Definizione dell'architettura del framework di interoperabilità domotica con analisi di vincoli ed interazioni con le LAN

Vittorio Miori (CNR) – Dario Russo (CNR)– Luca Ferrucci (CNR)

Breve sommario

In questo documento viene effettuata l'analisi delle interazioni tra il framework SHELL e le altre LAN, viste come le altre possibili installazioni domotiche di natura più o meno proprietaria (ad esempio, KNX, MyHome, Zwave, ZigBEE, ecc.). Dopo un breve escursus sulle caratteristiche di funzionamento dei framework di interoperabilità già esistenti in letteratura, ne vengono analizzate le problematiche generali in termini di limiti e vincoli possibili alla interoperabilità tra il framework SHELL e le LAN e tra le LAN connesse al framework stesso.

Parole chiave

Interoperabilità, LAN, MyHome, KNX, gateway

Indice

Indice delle Figure

Stato dell'arte

In questa sezione, vengono analizzate sia le modalità con cui operano alcuni tra i principali framework di interoperabilità esistenti e maggiormante diffusi a livello globale, sia come essi si comportano nella gestione delle interazioni con le altre LAN, cioè le reti domotiche preesistenti. I software in questione sono quelli vincolati da licenza GPL o freeware.

OpenHab

OpenHab è un framework di interoperabilità domotica creato nel 2010 da un consorzio di sviluppatori ed è diventato nel 2013 un progetto di Eclipse ufficiale sotto il nome di Eclipse Smarthome. E' completamente sviluppato in Java ed è basato su Eclipse Equinox il quale è a sua volta implementato seguendo le specifiche OSGi (Open Service Gateway initiative) sviluppate e definite dalla OSGi Alliance. La OSGi Alliance è un'organizzazione fondata nel 1999 da Ericsson, IBM, Oracle e altri. In seguito altri membri sono entrati a farne parte. Il nucleo delle specifiche è un middleware che definisce la gestione del modello del ciclo di vita del software, i moduli (chiamati *bundle*), un service registry e un ambiente di esecuzione. Partendo da questo middleware sono stati definiti un certo numero di OSGi Layer (strati), API e servizi.

Il framework OSGi si propone di implementare un modello a componenti completo e dinamico cioè quello che manca all'ambiente Java. In realtà il linguaggio Java prevede alcuni meccanismi che consentono di realizzare un modello a componenti, ma si tratta comunque di soluzioni di basso livello con conseguente rischio di introdurre errori nel codice oltre a diventare una soluzione ad-hoc. OSGi risolve molti dei problemi legati allo scarso supporto di Java nella modularità e nel dinamismo attraverso alcuni concetti fondamentali:

- Definizione del concetto di modulo (*bundle*)
- Gestione automatica delle dipendenze
- Gestione del ciclo di vita del codice (configurazione e distribuzione dinamica)

In sintesi è possibile vedere la tecnologia OSGi come:

- Un sistema modulare per la piattaforma Java
- Un sistema dinamico, che consente l'installazione, l'avvio, lo stop e la rimozione dei moduli a runtime, senza quindi necessitare di riavvii.
- Orientato ai servizi, i quali possono essere dinamicamente registrati ed utilizzati nella macchina virtuale Java.

Il Framework OSGi è distribuito su quattro layer:

- L0, Execution Environment (ambiente di esecuzione): è la specificazione dell'ambiente Java (J2SE, CDC, CLDC, MIDP, ecc.).
- L1, Modules: realizza il concetto di moduli (*bundle*) e controlla il collegamento tra di loro.
- L2, Life Cicle Management (Gestione del ciclo di vita): gestisce il ciclo di vita di un bundle senza richiedere il riavvio della VM.

- L3, Service Registry: fornisce un modello di cooperazione per i *bundle*.

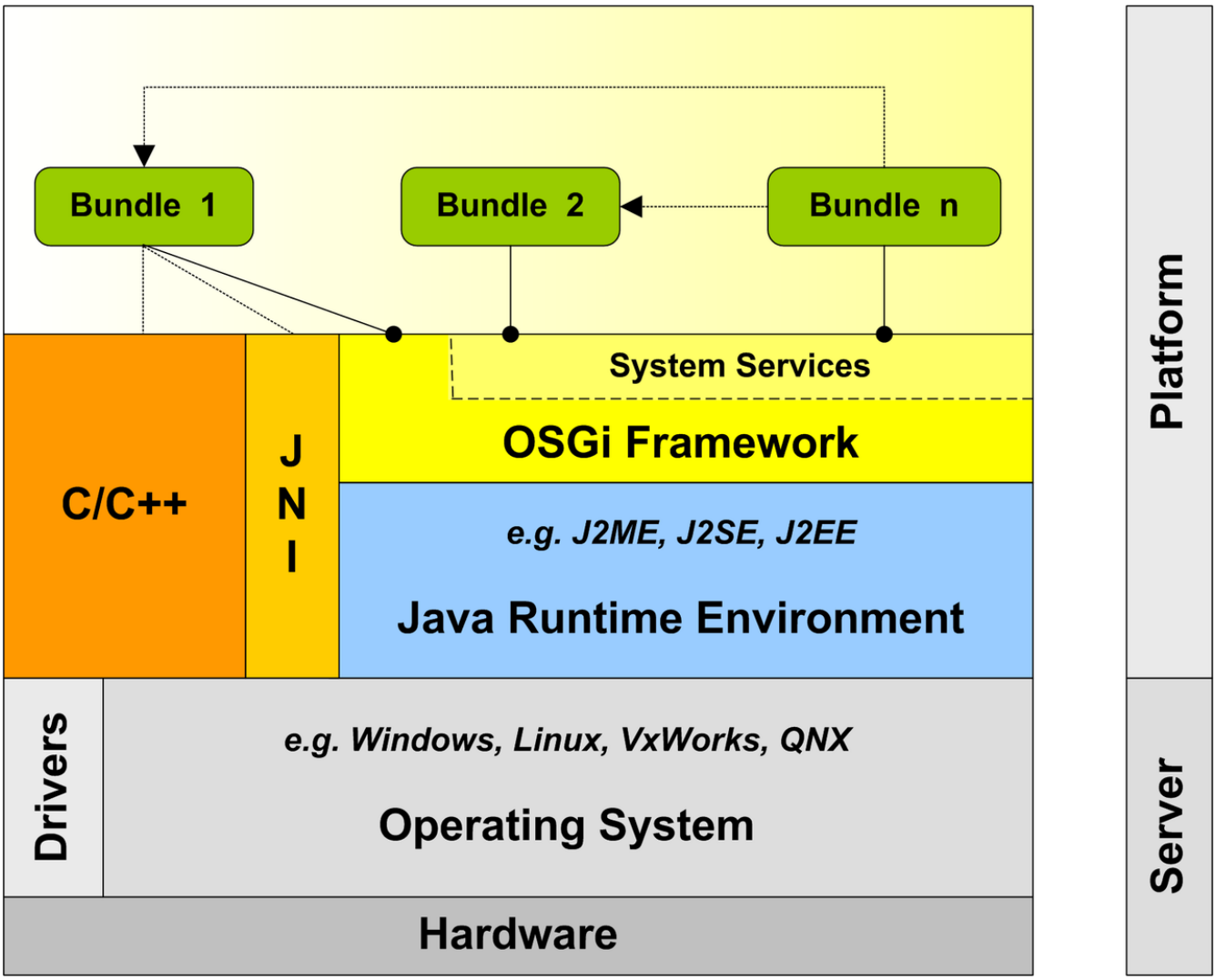

Figura 1: architettura middleware OSGi

Una configurazione di OpenHab si basa sui concetti di Things (*Cose*), Bindings (*Associazioni*), Channels (*Canali*) e Items (*Oggetti*). Le *Cose* rappresentano le singole fonti logiche di informazioni e funzionalità, come ad esempio i dispositivi domotici, web services, ecc… che devono essere gestiti dal sistema e rappresentano lo strato di entità fisiche collegate al framework tramite le *Associazioni*. Le *Associazioni* sono dei gateway di traduzione per specifici dispositivi. Per aggiungere una *Cosa* al framework, va identificata e caricata la necessaria *Associazione*. Ogni *Cosa* viene specificata tramite un apposito file di configurazione che può contenere ad esempio la seguente stringa in riferimento ad una webcam di sorveglianza posizionata in sala da pranzo, con indirizzo IPv4 192.168.0.2:

Thing network:device:webcam "Webcam" @ "Living Room" [hostname="192.168.0.2", timeout="5000", ...]

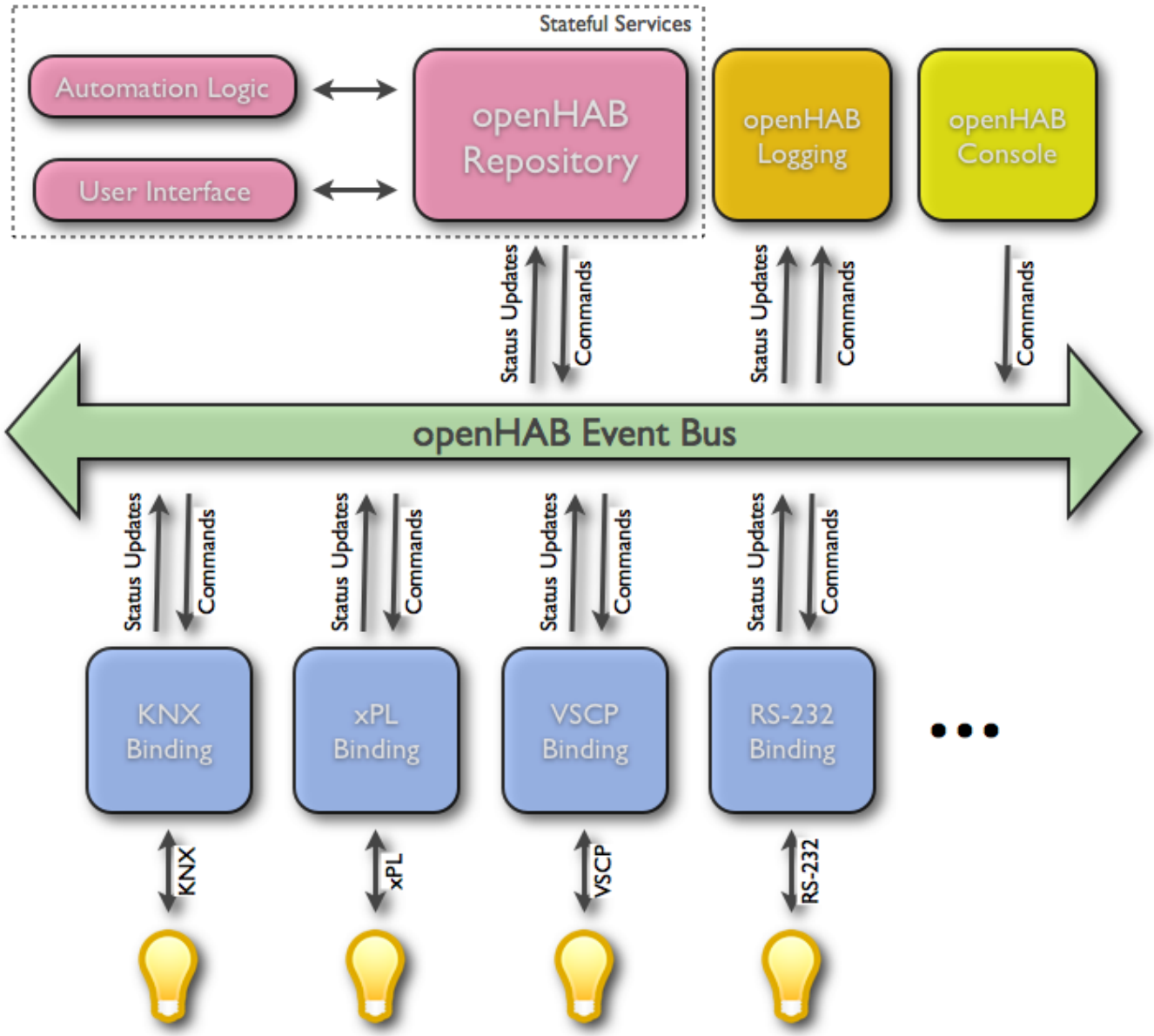

Figura 2: architettura di interoperabilità di openHAB

Una volta che una *Associazione* è stata caricata per una *Cosa*, essa crea uno o più *Canali* che sono collegati a un *Oggetto*, che rappresenta un blocco costruttivo atomico data-centrico, con stato, che può essere letto o scritto dalla *Cosa* (o *Cose*) a cui è collegato tramite uno o più *Canali*: il suo stato rappresenta lo stato del dispositivo fisico a cui l'*Oggetto* riferisce (in Figura 2 ad esempio le varie lampadine intelligenti).
OpenHAB definisce una tassonomia di tipi di *Oggetto* ognuno dei quali definisce i tipi di dati che possono essere letti o scritti e i comandi accettati, e definisce un insieme di messaggi ed eventi costituenti una "lingua franca" all'interno del framework: ogni Associazione ha il compito di convertire i messaggi ed eventi della specifica LAN in quelli interni al framework e viceversa, per poter scrivere e leggere lo stato degli *Oggetti* collegati alle *Cose*.
Per l'interoperabilità, in OpenHab è stato integrato un motore di regole che è in grado di pianificare o attivare una regola in base a diversi tipi di eventi e condizioni. Dopo aver attivato una regola, viene

invocato uno script per eseguire alcune operazioni o manipolazioni di elementi. Il linguaggio di scripting nativo è basato su framework Xbase e Xtend.

Freedomotic

Freedomotic è un *framework* di interoperabilità domotica creato nel 2013 e sviluppato in vari linguaggi, per la discovery, l'accesso e la gestione di dispositivi secondo il paradigma *dell'Internet of Things* (*IoT*, Internet delle Cose) e per l'automazione in generale, è multipiattaforma e pensato per la scalabilità.

Il progetto è nato per rispondere a una reale esigenza sorta nell'ambito di un programma di ricerca presso l'Università di Trento: occorreva un framework in grado di interconnettere vari progetti basati su reti di sensori, sviluppati in linguaggi di programmazione differenti e in tempi diversi, dando vita ad una piattaforma unificata per il testing, la valutazione e produzione di demo in grado di mostrare l'interazione di tali sistemi. In seguito, su iniziativa del suo fondatore Enrico Nicoletti, ha assunto una propria autonomia polarizzando intorno a sé una dinamica e appassionata comunità internazionale. E' stato il soggetto di numerosi lavori di tesi di laurea.

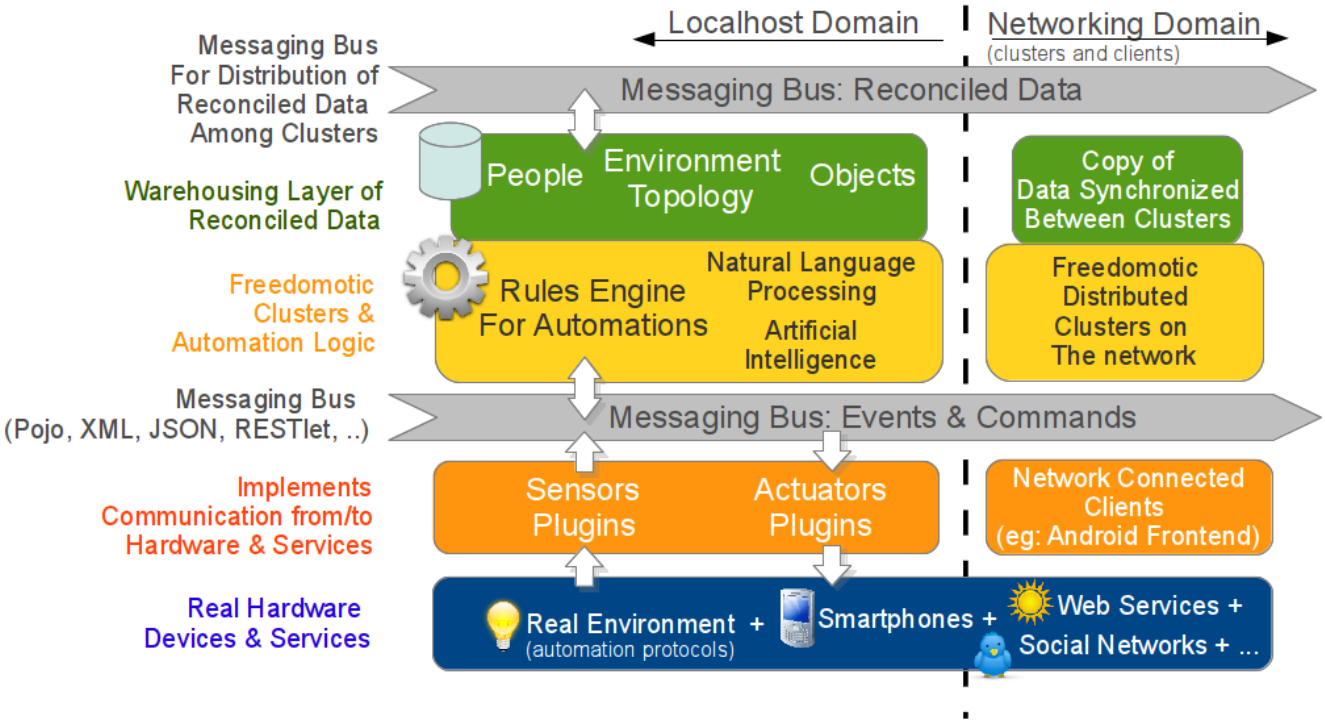

Figura 3: architettura di Freedomotic

L'architettura del sistema, mostrata in Figura 3, usa un *Bus Messaggi* che trasporta eventi e comandi basato sull'*Apache ActiveMQ Middleware*, su cui sono stati sviluppati tutti una serie di *plugin* che funzionano da *gateway* di traduzione tra il nucleo del *framework* e specifici dispositivi eterogenei collegati alle altre *LAN*. Il nucleo del *framework* svolge i seguenti compiti:

- Implementa un sistema di messaggistica indipendente dal linguaggio e basato *sull'Enterprise Integration Pattern*. Lo scopo del sistema di messaggistica è di far colloquiare fra loro in maniera interoperabile i vari plugins in un modo flessibile ed astratto, usando il concetto di *Canale*, basato su un sistema di publish-subscribe (pubblicazione/sottoscrizione) a differenti livelli di una gerarchia di argomenti.
- Mantiene una struttura dati interna che contiene la rappresentazione logica degli ambienti, degli oggetti nelle *Zone*, costituenti gli ambienti della casa, e il loro stato.
- Definisce un insieme di comandi e eventi costituenti una "lingua franca" all'interno del framework, allo stesso modo di *OpenHAB*: ogni plugin ha il compito di convertire i messaggi ed eventi della specifica *LAN* in quelli interni al *framework* e viceversa, e per poter definire una logica ad alto con cui poter descrivere, in *XML*, tutti i componenti del sistema (ambiente, oggetti, comandi, eventi e regole). Ad esempio, un comando ad alto livello potrebbe essere "accendi la luce in cucina".
- Fornire un insieme di regole ed un motore per la loro interpretazione in linguaggio naturale, come ad esempio "se fuori è scuro, accendi le luci della sala da pranzo".

I *plugin* sono divisi in due categorie, corrispondenti a due diversi livelli di astrazione:

- ***Plug-in* degli oggetti o *Cose***: modellano il comportamento degli oggetti del mondo reale (come le lampade e le porte), istruiscono il *framework* su quali comportamenti sono offerti da un oggetto e su quali operazioni supporta. Viene fornita una serie di comportamenti predefiniti, ma nuovi sviluppatori possono aggiungere nuovi tipi di *Cose*. I comportamenti hanno nomi che specificano il loro scopo (ad esempio il livello di luminosità) e un tipo, che specifica il tipo di funzione che implementano. Questo è correlato al tipo di parametro del controllo come, ad esempio, boolean, multivalore o intervallo di valori interi. Una cosa può essere virtuale o legata a un dispositivo reale, in questo caso deve essere contrassegnata come non virtuale e devono essre attribuiti un *protocollo* e un *indirizzo*. Il *protocollo* specifica il plug-in del driver da utilizzare, mentre l'*indirizzo* specifica come raggiungere il dispositivo a cui connettersi e la sua natura dipende dal protocollo: può essere ad esempio un indirizzo *IP* di un dispositivo di rete o un servizio web.
- ***Plug-in* dei dispositivi**: consentono di integrarsi con dispositivi hardware specifici o servizi specifici (come frontend grafici, servizi *Web*, motori di sintesi vocale e mittenti SMS). Per integrare nuovi protocolli di automazione domestica (es .: *ZWave*, *Zigbee*, *KNX* o *BTicino*) gli sviluppatori devono creare un plugin dedicato che funga da "traduttore" dai comandi generici *Freedomotic* ai comandi specifici del protocollo e viceversa. In pratica ciò significa connettere gli stati modificati del dispositivo hardware a comportamenti specifici delle *Cose* di *Freedomotic*.

Come altri tipi di framework di interoperabilità, *Freedomotic* non si basa su una ontologia o uno schema formalmente definiti, ma fornisce un modello liberamente strutturato di tipi di eventi e un insieme di proprietà associate e comportamenti di oggetti incorporati corrispondenti a modalità comuni di funzionamento di un dispositivo (ad es. accensione/spegnimento, livello di luminosità) e dei tipi di parametri possibili. Gli sviluppatori possono estendere il sistema di tipi di oggetto attraverso un'estensione *Java* e specificando e implementando le azioni fornite dal nuovo tipo.

Le regole di automazione, che definiscono le interazioni tra i dispositivi connessi alle varie *LAN* e quindi anche tra i *plugin*, sono basate sui concetti di *evento, trigger* e *comando.* Gli *eventi* (rappresentati dai canali) sono generati in modo programmatico dal dispositivo di gestione dei *plugin*. Un insieme di eventi comuni sono standardizzati all'interno della piattaforma utilizzando nomi gerarchici e nuovi possono essere creati dagli sviluppatori di *plugin*. Ogni evento ha una serie di proprietà che includono la *Cosa* che lo ha generato e i valori specifici dell'evento, come la temperatura associata a un evento di rilevamento. I *trigger* acquisiscono una serie di eventi che ne matchano i prerequisiti di attivazione. Possono semplicemente abbinare ad esempio un evento molto specifico a una persona specifica che si muove nella casa. Un componente può ascoltare tutti gli eventi notificati dai sensori nell'ambiente o implementare logiche più complesse esaminando le proprietà di un evento e utilizzando operatori logici e di confronto. I *comandi* sono azioni che devono essere eseguite da una singola o da una serie di *Cose* in risposta ai *trigger*. Contengono un insieme di proprietà che caratterizzano l'azione.

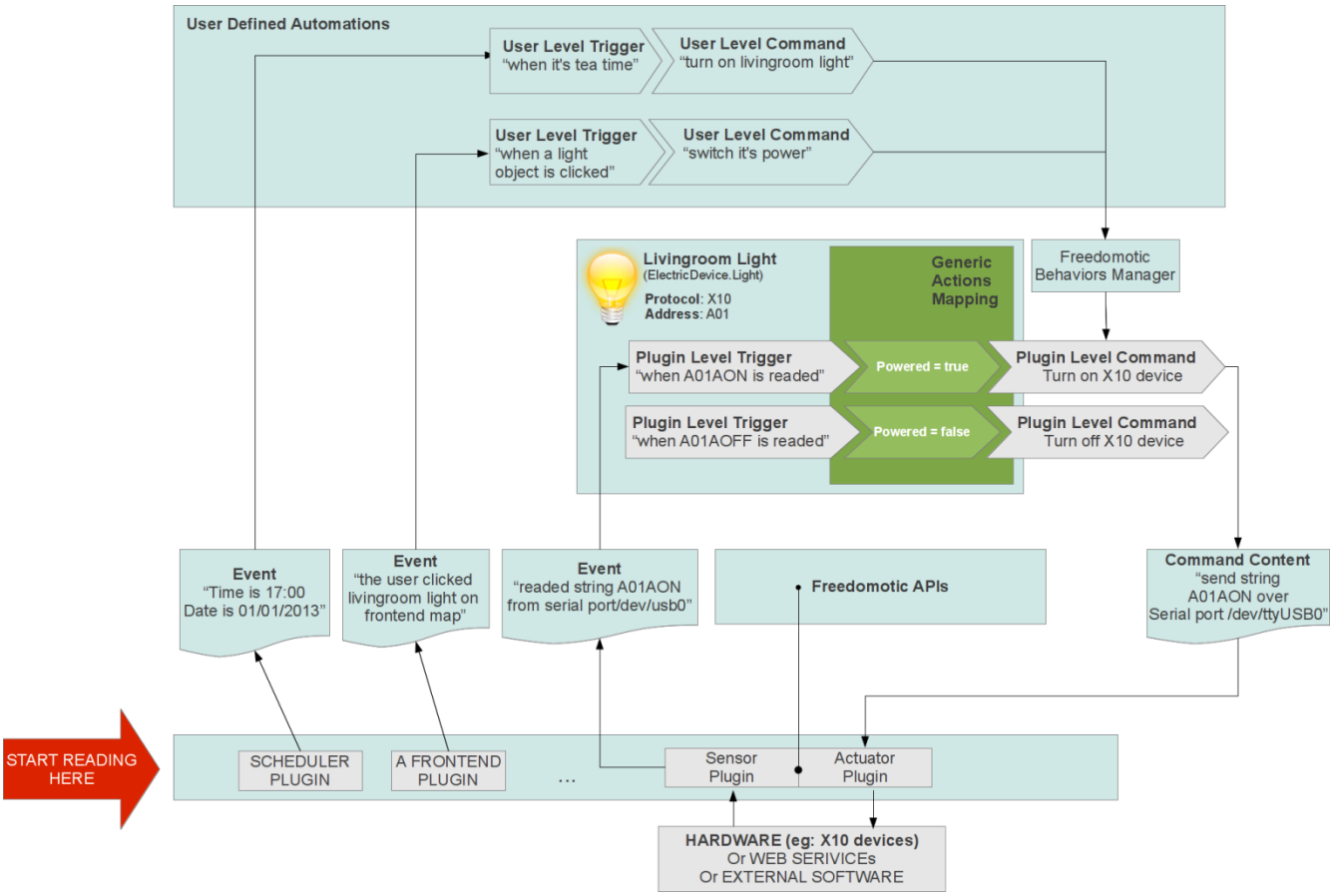

Figura 4: interoperabilità in *Freedomotic*

DomoNet

DomoNet è un *framework* di interoperabilità domotica sviluppato presso l'ISTI-CNR di Pisa nel 2006, con licenza *GPL*.

DomoNet considera la casa come un nodo *Internet* in grado di condividere ambienti e funzionalità dei dispositivi con qualsiasi altra istanza di *DomoNet* su *Internet*. Ad esempio, le funzionalità e gli stati

possono essere condivisi tra la residenza principale di un utente e la casa vacanze che implementa scenari di *IoT* reali.
L'interfaccia *DomoNet* è implementata utilizzando i servizi *Web* e le tecnologie *SOA*. Ciò consente a *DomoNet* di utilizzare approcci *W3C* standardizzati e riconosciuti per essere utilizzati e integrati da software di terze parti e altre istanze di *DomoNet*.
Ogni *LAN*, che rappresenta un sistema domotico preesistente, è rappresentata come una sottorete collegata al framework tramite un modulo dedicato chiamato *techManager*. I *techManager* sono *gateway* di traduzione che possono essere collegati a specifici dispositivi o a insiemi di dispositivi di una specifica *LAN* o tecnologia domotica, come ad esempio *KNX, BTicino, ZWave*, ecc… Per rappresentare dispositivi, servizi o eventi, *DomoNet* usa dei file di configurazione scritti in un linguaggio chiamato *DomoML*, un dialetto di *XML*, che usa XML schema e gioca il ruolo di un linguaggio comune per implementare l'interoperabilità. Esso è composto da due sotto-linguaggi:

- ***DomoDevice***: descrive le caratteristiche dei singoli dispositivi, la loro posizione nell'ambiente domotico, le funzioni disponibili denominate *servizi* ed il processo attraverso il quale l'interazione con gli altri servizi o dispositivi avviene e aggiunge una rappresentazione intermedia per i tipi di dati e la loro traduzione tra tecnologie domotiche diverse.
- ***DomoMessage***: standardizza il formato sintattico dei messaggi scambiati tra le rappresentazioni in *DomoNet* dei dispositivi, chiamate *DomoDevice*. Un messaggio ha diversi tipi possibili predefiniti.

Diversamente da altri *framework*, *DomoNet* non definisce una tassonomia predefinita dei messaggi o delle categorie dei dispositivi, quindi non possiede una "lingua franca" in cui i vari comandi, risposte o eventi delle varie *LAN* possano essere tradotti: il formato di un *DomoMessage* viene definito sintatticamente attraverso un file di configurazione in *DomoML* con un meccanismo quindi molto flessibile, ma il significato dei suoi campi è conosciuto solo dai *techManager* specifici, quindi non c'è alcun tipo di interoperabilità semantica. Il framework infatti possiede un set di regole specificabili attraverso un file di configurazione dove viene specificato il *mapping* tra dispositivi eterogenei: ad esempio, in Figura 6 viene mostrata l'interazione tra uno *switch* sulla *LAN KNX* ed una lampadina intelligente sulla *LAN X10*, con *ID DomoNet* di valore 10 e 3 rispettivamente. In Figura 5 è riportato il file di configurazione del dispositivo con ID 10: il file indica che il dispotivo, raggiungibile tramite l'interfaccia con url *http://www.thiswebservice.it/service* è uno *switch button*, che ha due *servizi*, *GetStatus* e *SetStatus* di tipo booleano, che servono rispettivamente a leggere e scrivere lo stato del bottone (premuto o non premuto); il tag *linkedService* indica che se viene invocato il servizio *SetStatus*, lo stato *status* viene passato al servizio *setPower* del dispositivo ID 3 (la luce sulla *LAN X10*), facendolo corrispondere allo stato *power* come indicato dal tag *linkedInput*: le conversioni fra i tipi vengono gestite dai due *techManager*, il *framework* e le regole si limitano ad effettuare il *routing* del *DomoMessage*.

```
<device description="bedroom switch button" id="10"
  url="http://www.thiswebservice.it/service"
  serialNumber="xyz" tech="KNX" type="switch button">
    <service output="Boolean" name="GetStatus"  />
    <service name="SetStatus">
```

```
            <input name="status" type="Boolean">
                        <allowed value="TRUE" />
                        <allowed value="FALSE" />
                </input>
        <linkedService id="3"
          url="http://www.otherwebservice.it/service" service="setPower">
                    <linkedInput from="status" to="power" />
            </linkedService>
        </service>
</device>
```

Figura 5: file di configurazione di uno switch su KNX

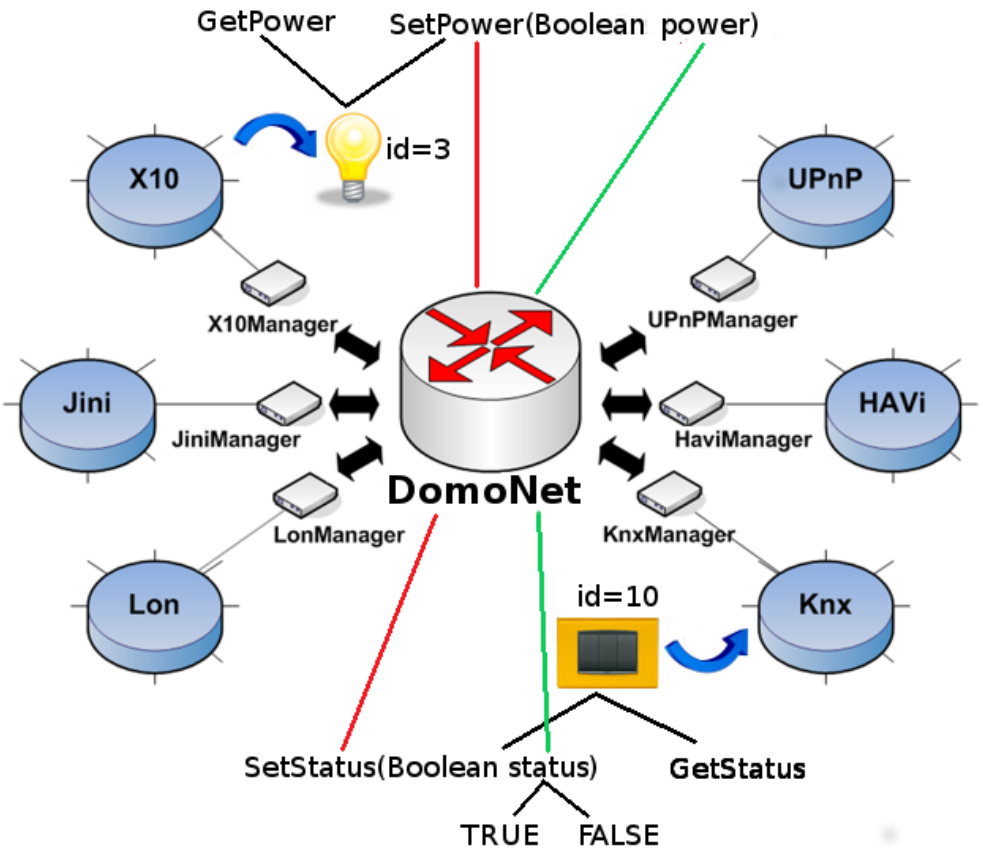

Figura 6: interoperabilità in DomoNet

Dog Gateway

Dog Gateway è un *framework* di interoperabilità domotica disegnato a partire da una ontologia ben conosciuta in ambito domotico, *DogOnt*, che viene utilizzata per descrivere strutturalmente un ambiente domotico intelligente. *Dog Gateway* è stato sviluppato per la prima volta nel 2006, principalmente con finalità di ricerca. Era mirato a una singola tecnologia di automazione domestica e aveva l'obiettivo di mostrare che un'interazione avanzata con gli ambienti domestici era possibile.

Questa prima versione fu chiamata *BCF*, dalle parole italiane “Basta Che Funzioni”, per evidenziare ulteriormente la sua natura transitoria.

Dal 2007 al 2014, *Dog Gateway* è stato sviluppato e migliorato incorporando più funzionalità, integrando nuove tecnologie di automazione domestica, edilizia e industriale, fornendo un unico livello di astrazione del dispositivo uniforme basato sull'ontologia *DogOnt*. In questi anni sono state rilasciate 3 versioni principali, la 3.0 è l'ultima.

Al giorno d'oggi, *Dog Gateway* è un'implementazione conforme *OSGi* di *gateway* di ambiente intelligente e piattaforma *IoT* con funzionalità multiprotocollo, *API* dell'applicazione basate su *REST*, *Uniform Device Representation* (Rappresentazione uniforme del dispositivo), attivazioni basate su regole, elaborazione dati in tempo reale (con complesse tecnologie di elaborazione degli eventi), integrazione con *Xively* (per consentire la condivisione sociale dei consumi e delle attivazioni), ecc.

DogOnt modella i dispositivi domotici organizzandoli in una tassonomia, specificando tutta una serie di classi di dispositivo, e fornisce costrutti per definire le loro funzionalità e le operazioni associate alle istanze di dispositivo (cioè ai singoli oggetti corrispondenti alle varie classi di dispositivo).
La peculiarità di *DogOnt* rispetto agli altri framework di interoperabilità domotica è di aver affrontato il problema partendo da una rappresentazione comune: l’ontologia infatti fornisce un modello di rappresentazione *cross-platform*, permettendo di rappresentare formalmente e univocamente i dettagli interoperazionali tra dispositivi eterogenei in modo da raggiungere una interoperabilità semantica completa.

L’ontologia rappresenta semanticamente i dispositivi, i loro stati e le funzionalità che possiedono: è scritta usando il *Web Ontology Language* (*OWL*), un linguaggio di *markup* per rappresentare esplicitamente significato e semantica di termini con vocabolari e relazioni tra gli stessi, estensione di *RDF* (*Resource Description Framework*). *RDF* è formalizzata tramite una parte decidibile della logica del primo ordine: ogni predicato è in relazione con altri predicati e permette di dichiarare l'esistenza di proprietà di un concetto, che permettano di esprimere con metodo sistematico affermazioni simili su risorse simili. *RDF* permette di definire il concetto di classe e sottoclasse, consente di definire gerarchie di classi, si possono rappresentare le risorse come istanze di classi e definire sottoclassi e tipi. L’ontologia può essere interrogata attraverso appositi reasoner logici in grado di interpretare *OWL*.

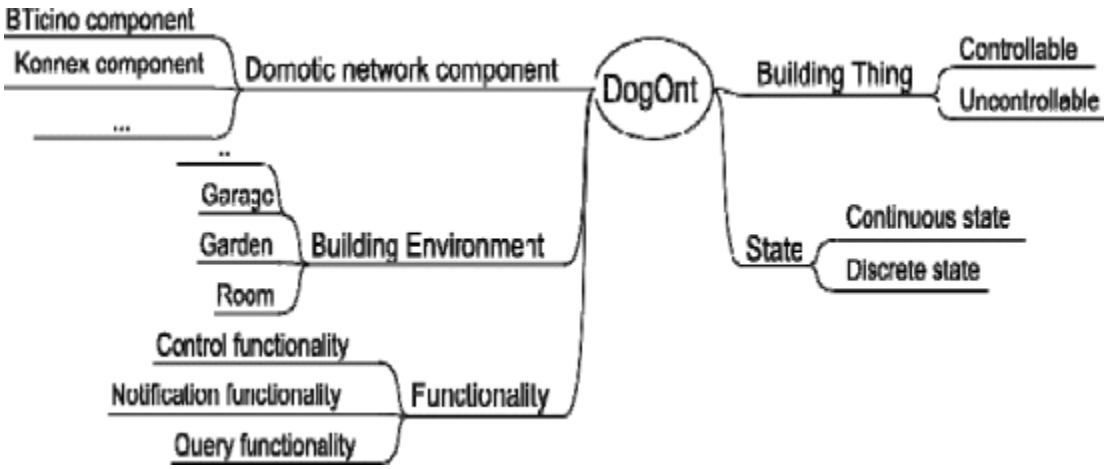

Figura 7: Dog ontology

L'ontologia viene sfruttata da *Dog Gateway* per separare nettamente la logica e l'implementazione, fornendo le funzionalità descritte dall'ontologia stessa per i vari dispositivi, potendo ad esempio validare i comandi inviati a run-time tramite opportune query alla base di conoscenza.

È organizzata su 5 alberi gerarchici principali (Figura 7), tra cui:

- **Building Thing**: modella gli elementi disponibili, cioè i vari dispositivi domotici e le categorie in cui si suddividono. Include il concetto di *Cosa controllabile* e i suoi discendenti, che sono usati per modellare dispositivi appartenenti a sistemi domotici o che possono essere controllati da essi.
- **Building Environment:** modella dove si trovano le cose, quindi i possibili ambienti della casa
- **State:** modella gli stati che possono assumere le cose controllabili
- **Functionality:** modella ciò che le cose controllabili possono fare
- **Domotic Network Component**: modella le caratteristiche peculiari di ogni impianto domotico (o LAN).

I dispositivi sono descritti in termini di *funzionalità* e le loro possibili configurazioni, cioè il loro *stato*. Le *funzionalità* sono principalmente suddivise in *continue* e *discrete*, le prime che descrivono capacità che possono essere variate continuamente e il secondo che si riferisce alla capacità di cambiare le configurazioni del dispositivo in modo discreto, ad esempio per accendere una luce. Inoltre, vengono classificati anche in base al loro obiettivo, ad esempio se consentono il controllo di un dispositivo (Funzionalità di *controllo*), l'interrogazione di una condizione del dispositivo (Funzionalità di *query*) o la notifica di una modifica di condizione (Funzionalità di *notifica*). Ogni istanza di *funzionalità* definisce l'insieme di comandi associati e, per le *funzionalità* continue, l'intervallo di valori consentiti, consentendo in tal modo la convalida a tempo di esecuzione dei comandi inviati ai dispositivi. I dispositivi possiedono anche un'istanza di stato derivante da una sottoclasse di *stato*, che descrive le configurazioni stabili che un dispositivo può assumere. Ogni classe di stato definisce l'insieme di valori di stato consentiti. La Figura 8 mostra la rappresentazione *DogOnt* di una lampada dimmer.

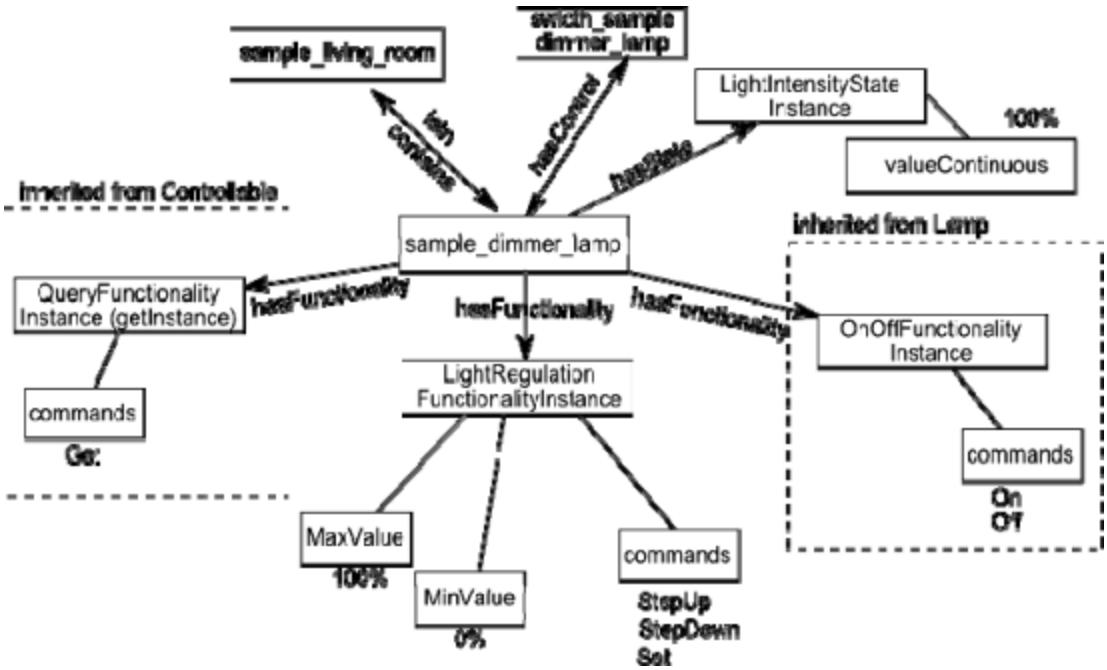

Figura 8: rappresentazione in DogOnt di un dimmer

I vari comandi, gli eventi e le funzionalità dei dispositivi descritte nell'ontologia vengono implementate attraverso il sistema fornito dal *framework OSGi* tramite un insieme di *DogMessages* definiti in *XML*, e tradotti da e per le *LAN* domotiche preesistenti, come *KNX o BTicino*, tramite una serie di *driver* software che sfruttano la base di conoscenza per la traduzione e il mapping tra i dispositivi connessi.

DOG è organizzato in un'architettura a strati con 4 anelli, ognuno dei quali si occupa di diversi compiti e obiettivi, che vanno da problemi di interconnessione di basso livello a modellazione e interfaccia di alto livello. Ogni anello include diversi *bundle OSGi*, corrispondenti ai moduli funzionali della piattaforma:

- **Anello 0**: include la libreria comune *DOG* e i bundle necessari per controllare e gestire le interazioni tra la piattaforma *OSGi* e gli altri *bundle DOG*. A questo livello, gli eventi di sistema relativi a configurazioni di runtime, errori o guasti, vengono generati e inoltrati all'intera piattaforma *DOG*.
- **Anello 1**: comprende i pacchetti *DOG* che forniscono un'interfaccia alle varie reti domotiche o *LAN* a cui è possibile collegare il *framework*. Ogni *LAN* è gestita da un driver dedicato che funge da *gateway*.
- **Anello 2**: fornisce l'infrastruttura di routing per i messaggi che viaggiano attraverso i *driver* di rete e diretti ai *bundle DOG*. L'anello 2 ospita anche l'intelligenza di base di *DOG*, basata sull'ontologia *DogOnt*, implementata nel bundle *House Model*.
- **Anello 3:** ospita i pacchetti *DOG* che consentono l'accesso ad applicazioni esterne, tramite un *bundle API*, per applicazioni *OSGi* o tramite un *endpoint XML-RPC* per applicazioni basate su altre tecnologie.

Per interfacciare le altre *LAN* o tecnologie domotiche, *DOG* fornisce un set di driver di rete, uno per ogni tecnologia diversa (ad esempio, *KNX*, *OpenWebNet* per *BTicino*, ecc…), facenti parte dell'Anello 1. Ogni

driver di rete implementa una fase di "auto-configurazione", in cui interagisce con il modello della casa domotica (attraverso l'interfaccia di *HouseModeling*) per recuperare l'elenco dei dispositivi da gestire, insieme a una descrizione delle loro funzionalità, nel formato *DogOnt*. Ogni descrizione del dispositivo porta tutte le informazioni di basso livello necessarie come l'indirizzo del dispositivo, in base al formato di indirizzamento dipendente dalla rete (semplice in *OpenWebNet*, suddiviso in gruppi e singoli indirizzi in *KNX*, ecc.). I driver di rete traducono i messaggi avanti e indietro tra i pacchetti *Dog* e i *gateway* a livello di rete e implementano un'interfaccia che supporta *query* e comandi emessi da altri *gateway DOG*. Inoltre, utilizzano i servizi definiti da un'interfaccia chiamata *StatusListener* per propagare le modifiche di stato ai moduli software registrati. Il monitoraggio, a livello di driver di rete, può essere eseguito ascoltando gli eventi di rete o eseguendo cicli di *polling*. In entrambi i casi, i *bundle* dei *driver* forniscono aggiornamenti di stato agli altri *bundle DOG* utilizzando il tipico paradigma d'interazione basato sugli eventi supportato da *OSGi*. Attualmente sono già stati sviluppati tre driver di rete: *Konnex, OpenWebNet* e *Simulator*. I driver *KNX* e *BTicino* traducono *DogMessages* in messaggi di protocollo delle reti corrispondenti. Il driver del simulatore, invece, emula un ambiente sintetico generando eventi casuali o riproducendo tracce di eventi registrati (ciò consente di simulare situazioni critiche in un ambiente sicuro).
Ogni *DogMessage* si basa sulla rappresentazione *DogOnt* dei dispositivi (reali o virtuali), che è indipendente dalle tecnologie sottostanti. Ciò consente di eseguire chiamate *API* indipendenti dalla tecnologia che vengono quindi instradate dal bundle *Message Dispatcher* al driver di rete corretto. L'interoperabilità può essere di due tipi principali: da rete a rete e da applicazione a rete. Nel primo caso, i dispositivi appartenenti ad una data *LAN*, ad esempio uno *switch KNX*, comunicano con dispositivi collegati ad un'altra rete, ad es. una lampada *OpenWebNet*. Il secondo caso invece è applicato ogni volta che un'applicazione (sia basata su *OSGi* o *XML-RPC*) deve interagire con dispositivi appartenenti a sistemi domotici diversi allo stesso tempo. Attualmente, *Dog* supporta l'interoperabilità tra applicazioni sfruttando le *query SPARQL* sull'ontologia e la generazione automatica di *DogMessages*.

La difficoltà maggiore nel modello di *DOG* è la rigidità di mantenere un'ontologia scritta in un linguaggio formale: l'estensione della stessa è difficoltosa e necessità di strumenti adatti come *Protégé*, e se non fatta correttamente, può portare a una base di conoscenza incompleta o contraddittoria.

Home Assistant

Home Assistant è un *framework* di interoperabilità domotica creato da una comunità *open source* fondata da *Paulus Shoutsen*, basato su *Python 3* in grado di controllare tutti i dispositivi domotici all'interno di una casa. Ogni versione introduce funzionalità sempre più utili e integra sempre più dispositivi da controllare. Oggi ha 583 *componenti* e piattaforme che integrano dispositivi e servizi come *Amazon Eco, Apple TV, Arduino, IKEA Tradfri, Philips Hue, Z-Wave, KNX* e così via.
Home Assistant è controllabile da qualsiasi dispositivo che supporti un browser Internet. Android, PC, Apple ecc. Una volta installato, il *frontend* è accessibile da un *server web*. In Figura 9 si può vedere l'architettura di *Home Assistant*.

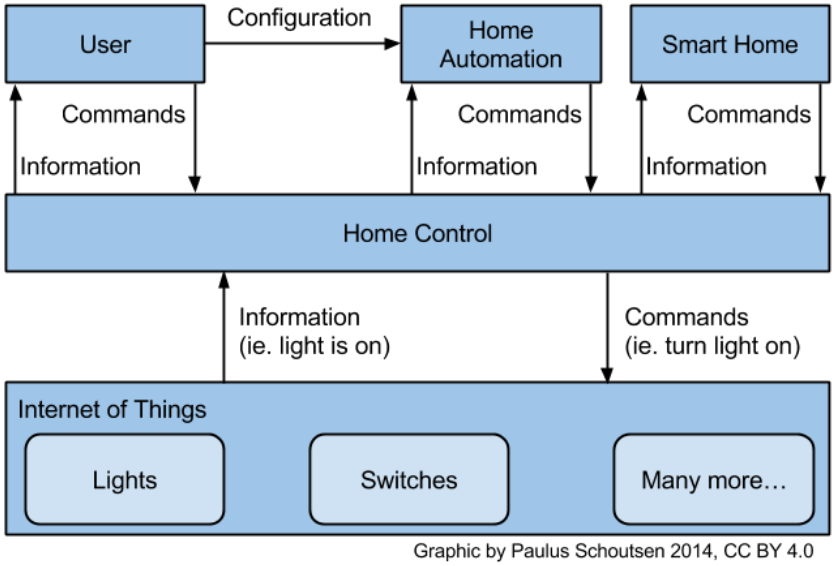

Figura 9: Architettura di Home Assistant

Home Control è responsabile della raccolta delle informazioni e del controllo dei dispositivi. I comandi ai dispositivi possono essere avviati da *trigger* definiti nel modulo *Domotica* tramite configurazioni utente e dal modulo *Smart Home* tramite la conoscenza e la raccolta d'informazioni sul comportamento precedente dell'utente.
Home Control è composto da cinque parti:

- **Bus eventi**: facilita lo sparo e l'ascolto degli eventi, il cuore pulsante di *Home Assistant*.
- **State Machine**: tiene traccia degli stati delle cose e genera un evento *state_changed* quando uno stato è stato modificato.
- **Registro servizi**: ascolta il bus eventi per gli eventi *call_service* e consente ad altri codici di registrare i servizi.
- **Timer**: invia un evento di cambio del tempo ogni secondo sul bus eventi.
- **Componenti** della casa, come ad esempio i dispositivi fisici domotici.

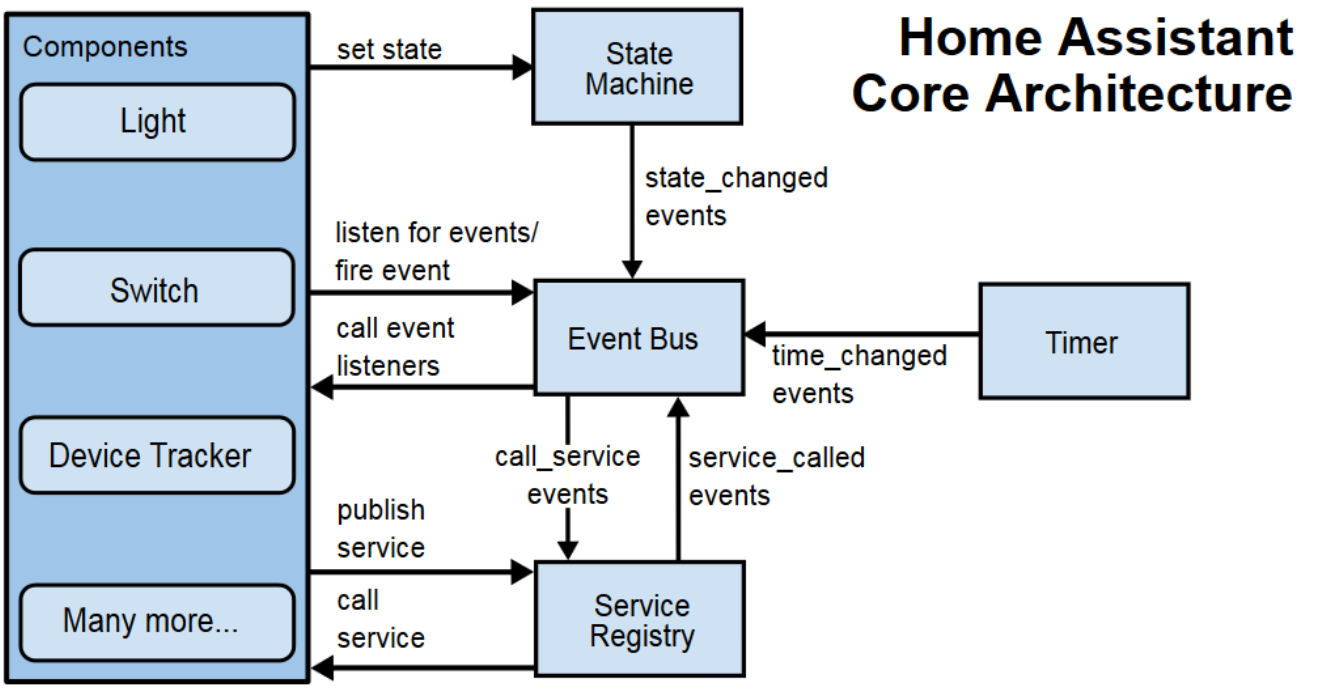

Figura 10: Architettura del core di Home Assistant

Home Assistant può essere esteso con *componenti*. Ogni *componente* è responsabile di fungere da gateway di traduzione con una certa tecnologia domotica o *LAN* all'interno di *Home Assistant*. I *componenti* possono ascoltare o attivare eventi, offrire servizi e mantenere gli stati dei dispositivi collegati a quella *LAN* domotica. I *componenti* sono scritti in *Python* e ve ne sono un certo numero di predefiniti.

Esistono due tipi di *componenti* all'interno di *Home Assistant*: *componenti* che interagiscono con un dominio *IoT* e le altre *LAN* domotiche e componenti che rispondono agli eventi che si verificano all'interno di *Home Assistant*, il quale consente di definire regole di automazione specifiche.

L'interoperabilità è ottenuta disaccoppiando i dispositivi (ad esempio interruttori, luci, ecc.) dalla logica specifica della piattaforma che definisce come interagire con dispositivi specifici e protocolli domotici esistenti. Alcune piattaforme sono già state integrate, come quella che gestisce i dispositivi *Z-wave* e *KNX* ma possono essere aggiunte di più dagli sviluppatori implementando metodi per gestire comandi specifici della piattaforma e leggere stati dei dispositivi.
I *componenti* interagiscono quindi con il sistema attivando e consumando eventi, fornendo così un livello di astrazione e in definitiva consentendo l'interoperabilità tra i dispositivi che implementano nativamente diversi standard domotici.
Gli sviluppatori che desiderano aggiungere il supporto per i nuovi dispositivi possono scrivere la logica della piattaforma solo riutilizzando la logica di base per quel tipo specifico di dispositivo. Ad esempio, un *KNXSwitch* eredita le logiche di base dall'entità *SwitchDevice* generica, implementa metodi specifici per attivare e consumare eventi ed eventualmente estendere le funzionalità di base fornite dalla sua classe genitore.

La piattaforma specifica associata a un dispositivo può essere specificata nel file di configurazione di *Home Assistant* ed eventualmente modificata in un secondo momento per sostituire il dispositivo con un altro dello stesso tipo ma che implementa nativamente un diverso protocollo domotico.
Il *bus* eventi di *Home Assistant* consente a qualsiasi *componente* di attivare o ascoltare eventi. Ad esempio, qualsiasi variazione di stato viene annunciata sul *bus* eventi come evento *stato_cambiato* contenente il precedente e il nuovo stato di un'entità. E' un *componente* apposito, il ***trigger***, ad avviare successivamente l'elaborazione di una regola di automazione. Una volta avviato il *trigger*, *Home Assistant* convaliderà le condizioni, se presenti, e invocherà ed eseguirà l'azione.

IoTivity

IoTivity è un framework di interoperabilità domotica creato sulla base delle specifiche *Open Connectivity Foundation* (OCF) create dal consorzio *OCF* in uno sforzo comune di standardizzazione per la discovery, l'accesso e la gestione di dispositivi secondo il paradigma *IoT*. Le specifiche *OCF* sono state create per essere agnostiche rispetto al dominio di applicazione, quindi il dominio domotico è una delle tante implementazioni delle specifiche possibile, in quanto le stesse definiscono il modello generico delle Risorse, dell'accesso alle stesse attraverso un'interfaccia standard, della discovery, della presentazione dei servizi, della comunicazione e della sicurezza. I membri del consorzio includono *Cisco*, *Samsung*, *Intel*, *Qualcomm*, *Mediatek* e altri. L'obiettivo di *IoTivity* è quello di creare un nuovo standard commerciale che crei un'architettura robusta ed estensibile per dispositivi intelligenti. *IoTivity* consente comunicazioni da dispositivo a dispositivo e da dispositivo a Internet supportando diversi stack di protocollo come *BLE*, *NFC*, *BlueTooth*, *IPv4 / IPv6*.

Il framework *Iotivity* è costituito da funzioni che forniscono funzionalità di base per supportare varie operazioni, tra cui il *Modello di Risorsa*, il quale fornisce le astrazioni e i concetti necessari per modellare logicamente l'applicazione e il suo ambiente. Il *Modello di Risorse* di base, definito nelle specifiche *OCF*, è comune e indipendente da qualsiasi dominio applicativo specifico, come nello stesso modo delle operazioni, fornendo coerenza e interoperabilità tra i dispositivi.
I concetti e i meccanismi del *Modello di Risorsa* vengono quindi mappati sui protocolli di trasporto per consentire la comunicazione tra i dispositivi (ogni protocollo di trasporto fornisce l'interoperabilità del protocollo di comunicazione) in modo tale che essa sia indipendente dal livello trasporto stesso. Inoltre, i concetti nel modello di risorse supportano la modellazione degli artefatti primari e le loro relazioni con l'uno e l'altro e acquisiscono le informazioni semantiche necessarie per l'interoperabilità in un contesto.

Le specifiche OCF introducono i concetti di *Entità*, *Risorse*, *URI*, *Tipi di risorsa*, *Proprietà*, *Rappresentazioni*, *Interfacce*, *Collezioni* e *Collegamenti*. Questi concetti e meccanismi possono essere composti in vari modi per definire la ricca semantica e l'interoperabilità necessarie per una serie diversificata di casi d'uso a cui il framework *OCF* è applicato. La *Risorsa* è la rappresentazione di una *Entità*, cioè di un dispositivo fisico che necessita di visibilità, di interagire e si essere manipolato, e la rappresentazione di una risorsa è accedibile attraverso interazioni e operazioni definite secondo lo stile architetturale definito *Representational State Transfer* (REST). La *Risorsa* incapsula lo stato dell'*Entità*,

ed è identificata, indirizzata e nominata attraverso il ben noto standard di accesso *URI,* ed è gestita da un dispositivo. Un *URI* OCF si basa sulla forma generale di un *URI* come definito nel *RFC 3986*:

```
<Schema>: // <Autorità> / <Percorso>?<Query>
```

Nello specifico l'URI *OCF* è specificato nel seguente formato:

```
OCF: // <Autorità> / <Percorso>?<Query>
```

Lo schema 'ocf' rappresenta la semantica e le definizioni definite nelle specifiche OFC. L'*Autorità* di un URI *OCF* è il valore *Device ID* del Server. Il *Percorso* è una stringa che identifica in modo univoco o fa riferimento a una risorsa nel contesto del server, mentre la *Query* deve contenere un elenco di coppie "nome-valore", che verranno mappate alla sintassi appropriata del protocollo utilizzato per la messaggistica (ad es. *COAP*). Le *Proprietà* di una risorsa sono coppie di tipo Chiave-Valore, e il loro valore in un certo istante di tempo è la sua *Rappresentazione.* Un'istanza di *Risorsa* deriva da un *Tipo di risorsa*. La relazione unidirezionale tra una *Risorsa* e un'altra *Risorsa* è definita come un C*ollegamento*. Una *Risorsa* che ha uno o più riferimenti (proprietà e collegamenti) ad altre risorse è una *Collezione*. Questi riferimenti possono essere correlati tra loro o semplicemente essere una lista; la *Collzione* fornisce un mezzo per riferirsi a questa serie di riferimenti con un singolo *URI*. Una semplice *Risorsa* viene mantenuta distinta da una *Collezione*: qualsiasi *Risorsa* può essere trasformata in una *Collezione* legando i riferimenti alle risorse come collegamenti. Le *Collezioni* possono essere utilizzate per creare, definire o specificare gerarchie, indici, gruppi e così via. Un insieme di proprietà può essere utilizzato per definire uno stato di una *Risorsa*. Questo stato può essere recuperato o aggiornato utilizzando rappresentazioni appropriate rispettivamente nella risposta e richiesta a quella risorsa. Una *Risorsa* (e un *Tipo di risorsa*) e le interazioni con tale *Risorsa* possono rappresentare ed essere utilizzata per esporre una capacità. Tali capacità possono essere utilizzate per definire processi come scoperta, gestione, pubblicità, ecc. Ad esempio: "scoperta di *Risorse* su un dispositivo" può essere definita come il recupero di una rappresentazione di una *Risorsa* specifica in cui una proprietà o proprietà hanno valori che descrivono o fare riferimento alle *Risorse* sul dispositivo. Le informazioni per la richiesta o la risposta con la *Rappresentazione* possono essere comunicate utilizzando un protocollo di trasferimento.

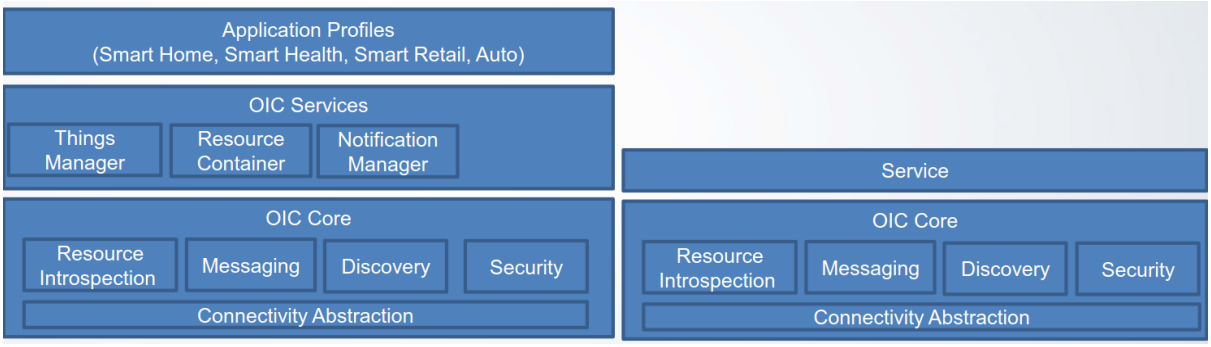

Figura 11: Architettura del core di IoTivity

Le specifiche OCF prevedono un set predeterminato di operazioni generiche di accesso allo stato di una *Risorsa* indipendenti dal protocollo di trasmissione e rappresentazione utilizzato. Ogni *Risorsa* può agire indifferentemente come *Iniziatore* (Client) o *Risponditore* (Server) per un'operazione, agendo seguendo

determinati ruoli di volta in volta assegnati: un dispositivo può ad esempio assumere il ruolo di server se espone un insieme di *Risorse* a cui accedere pubblicamente attraverso un'*Interfaccia*.
Le operazioni predefinite sono le seguenti e sono chiamate ***CRUDN***:

- **Creazione**: il messaggio richiede al server di creare una nuova *Risorsa* sulla base della *Rappresentazione* contenuta nel messaggio stesso.
- **Recupero**: il messaggio richiede al server di rispondere con il corrente stato o la *Rappresentazione* di una *Risorsa* con *URI* specificato nel messaggio stesso
- **Aggiornamento**: il messaggio richiede al server di aggiornare (parzialmente o completamente) l'informazione in una *Risorsa* con *URI* e nuove proprietà specificati nel messaggio stesso, secondo le regole dell'*Interfaccia* implementata dal server per la specifica *Risorsa*
- **Cancellazione**: il messaggio richiede al server di rimuovere o cancellare la *Risorsa* con *URI* specificato nel messaggio stesso
- **Notifica**: il messaggio richiede al server di notificare, in maniera asincrona, il cambiamento di stato della *Risorsa* con *URI* specificato nel messaggio stesso.

La configurazione di un dispositivo è dinamica e ogni dispositivo può connettersi / disconnettersi autonomamente dalla struttura tramite un meccanismo di registrazione. Ogni dispositivo *OCF* deve associarsi ad almeno un endpoint con il quale può scambiare messaggi di richiesta e risposta. Quando un messaggio viene inviato a un endpoint, deve essere consegnato al dispositivo OCF associato all'endpoint. Quando un messaggio di richiesta viene consegnato a un endpoint, il componente *Percorso* dell'*URI* è sufficiente per individuare la *Risorsa* di destinazione. Il dispositivo *OCF* può essere associato a più endpoint: ad esempio, un dispositivo *OCF* può avere diversi indirizzi *IP* o numeri di porta o supportare entrambi i protocolli di trasferimento *CoAP* e *HTTP*. L'endpoint è rappresentato da una coppia chiave-valore, "ep":

```
"ep": "coap: // [fe80 :: b1d6]: 1111"
```

"Ep" rappresenta *Transport Protocol Suite* ed *Endpoint Locator*, dove il primo è una combinazione di protocolli (ad es. *CoAP* + *UDP* + *IPv6*), mentre il secondo è un indirizzo (es. Indirizzo *IPv6* + numero di porta) attraverso il quale un messaggio può essere inviato a l'endpoint e a sua volta ad un dispositivo associato. Un endpoint potrebbe essere scoperto implicitamente o esplicitamente; nel primo caso, se un dispositivo è la fonte di un messaggio *CoAP* (ad esempio, risposta "/oic /res"), l'indirizzo *IP* di origine e il numero di porta possono essere combinati per formare il localizzatore di endpoint per il dispositivo. Altrimenti, quando l'endpoint per una *Risorsa* di destinazione di un *Collegamento* non può essere dedotto implicitamente, il parametro "eps" deve essere incluso per fornire informazioni esplicite sull'endpoint con cui un client può accedere alla risorsa di destinazione.

Functionality	Description
Discovery	IoTivity discovery supports multiple discovery mechanisms for devices and resources in proximity and remotely
Data Transmission	IoTivity data transmission supports information exchange and control based on a messaging and streaming model
Data Management	IoTivity data management supports the collection, storage and analysis of data from various resources.
Device Management	IoTivity device management supports configuration, provisioning and diagnostics of devices.

Figura 12: funzionalità principali di IoTivity

In base alla scelta degli aspetti di base, *OCF* definisce tre meccanismi di scoperta basati sulle *Risorse*:

- **Scoperta diretta**, in cui le *Risorse* sono pubblicate localmente sul dispositivo che ospita le *Risorse* e vengono scoperte tramite l'indagine tra pari.
- **Scoperta indiretta**, in cui le *Risorse* vengono pubblicate da terze parti che assistono la scoperta e i peer pubblicano ed eseguono la scoperta della *Risorsa*.
- **Scoperta basata su pubblicazione**: in cui la *Risorsa* per abilitare la scoperta viene ospitata localmente per l'iniziatore dell'indagine di individuazione, ma remota ai dispositivi che pubblicano le informazioni di individuazione.

L'interoperabilità tra i dispositivi e le *Risorse IoTivity* si basa sulle specifiche *Bridging OCF*, che specificano come interconnettere le implementazioni *OCF* e *no-OCF*. L'idea è basata su dispositivi *OCF Bridge*, che possono rappresentare dispositivi che esistono sulla rete ma comunicano usando un protocollo *Bridged* piuttosto che i protocolli *OCF*. Il dispositivo *Bridge* funziona come un *gateway*: espone ogni *client* o *server Bridged* (un oggetto *no-OCF*) in un *client* o *server OCF* virtuale, che lo rappresenta logicamente. Dall'altro lato, il *server* e il *client* virtuali *Bridged* sono rappresentazioni logiche di server e client *OCF*, che un dispositivo *Bridge OCF* espone al *server* e ai *client Bridged*.
Il tipo di traduzione principale della specifica di *bridging OCF* è la **traduzione profonda**, che è una traduzione in cui i modelli di dati utilizzati con il protocollo *Bridged* sono mappati ai tipi di risorse *OCF* equivalenti e viceversa, in modo tale che un client *OCF* o un *client Bridged* conforme possa essere in grado di interagire con il servizio in modo completamente trasparente.
Un dispositivo *Bridge OCF* è un dispositivo che rappresenta uno o più dispositivi *Bridged* come dispositivi *OCF* virtuali sulla rete e/o rappresenta uno o più dispositivi *OCF* come dispositivi virtuali per una rete *no-OCF*. L'unica differenza tra un dispositivo *OCF* nativo e un dispositivo con ponte virtuale è come il dispositivo è incapsulato in un dispositivo *Bridge OCF*. Un dispositivo *Bridge OCF* deve essere indicato sulla rete *OCF* con un tipo di dispositivo "oic.d.bridge", per fornire a un client *OCF* un'indicazione esplicita che il dispositivo rilevato sta eseguendo la funzione di bridging. Quando viene scoperto un dispositivo di questo tipo, le risorse esposte sul dispositivo *Bridge OCF* descrivono altri dispositivi, come mostrato in Figura 13: 3 dispositivi domotici su una *LAN* esterna (due luci e una ventola a velocità variabile) sono rappresentati dalle tre risorse virtuali con URI *oic.d.fan*, *oic.d.light1* e *oic.d.light2* rispettivamente, esposte da un *OCF Bridge Device*. Solitamente, il dispositivo *Bridge OCF* crea l'insieme di dispositivi virtuali durante l'avvio: questi possono cambiare in quanto i dispositivi Bridged possono

essere aggiunti o rimossi dinamicamente dal bridge. Quando un dispositivo *Bridge OCF* cambia il set di dispositivi virtuali esposti, deve notificarlo a tutti i *client OCF* iscritti al suo "/oic/ res".
Il dispositivo *Bridge OCF* è coinvolto anche nel processo di scoperta delle *Risorse*: rileva i dispositivi che arrivano ed escono dalla rete *Bridged* o dalla rete *OCF*. Laddove non esiste un meccanismo preesistente per rilevare in modo affidabile l'arrivo e la partenza dei dispositivi su una rete, un dispositivo *OCF Bridge* deve periodicamente eseguire il *polling* della rete per rilevare l'arrivo e la partenza dei dispositivi, ad esempio utilizzando il rilevamento *multicast* di *COAP* (un *multicast RETRIEVE* di "/oic/res") nel caso della rete *OCF*. Un dispositivo *Bridge OCF* deve rispondere ai comandi di rilevamento della rete per conto dei dispositivi a ponte esposti, come ad esempio la risposta a *RETRIEVE* su "/oic/res" deve includere solo i dispositivi che corrispondono alla richiesta *RETRIEVE* e inviati dal dispositivo *Bridge OCF* per conto di dispositivi virtuali. Inoltre, un dispositivo *Bridge OCF* deve controllare l'UUID indipendente dal protocollo di qualsiasi dispositivo no-OCF, che è chiamato "piid" nelle specifiche *OCF*, e il tipo di dispositivo univoco "oic.d.virtual" di qualsiasi dispositivo *OCF*, per prevenire più pubblicazioni e/o simulazioni della stessa risorsa virtuale (sul lato *OCF* o *no-OCF* della rete). Infine, ogni server *Bridged* è esposto come un server virtuale *OCF* separato, con un proprio endpoint e una directory di risorse, e ogni dispositivo virtuale deve implementare i requisiti di sicurezza dell'ecosistema al quale è connesso: ad esempio, ogni dispositivo *Virtual OCF* deve implementare il comportamento richiesto dalla sicurezza *OCF* e dalle specifiche principali.

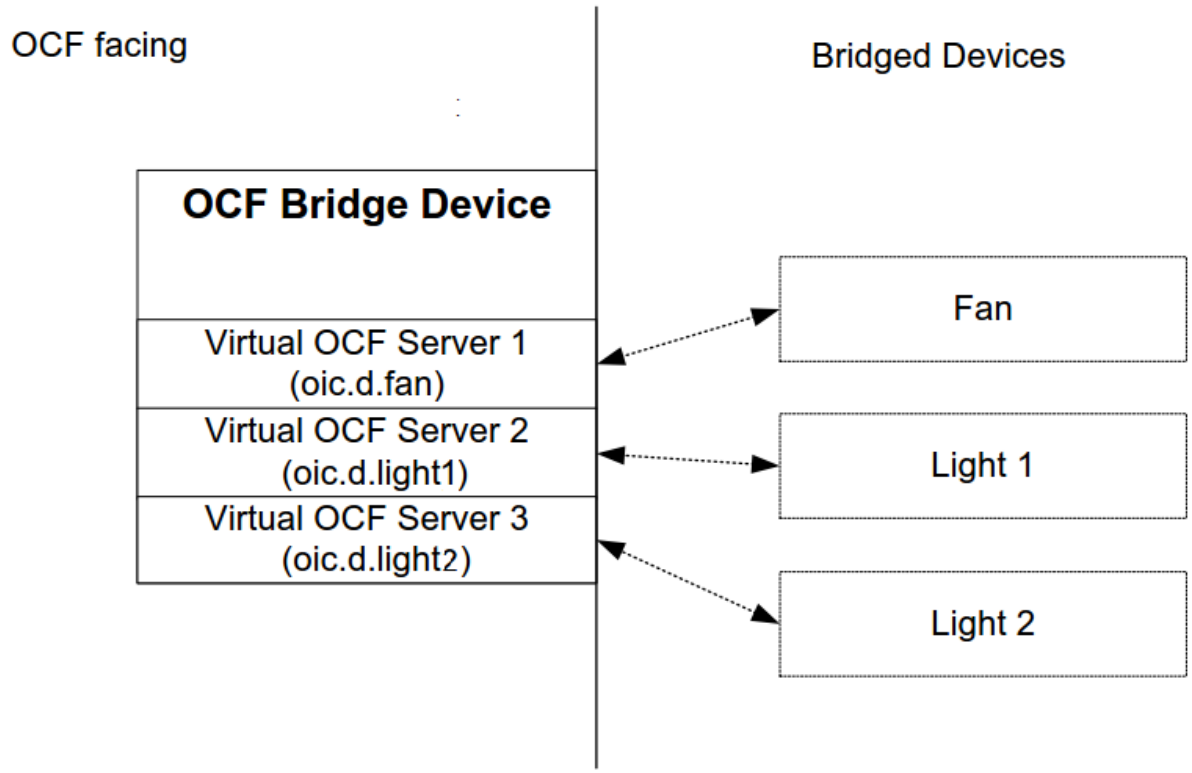

Figura 13: Risorsa OCF Bridge e interazione con i dispositivi esterni

IoTSys è un framework di interoperabilità domotica per l'*IoT* costituito da uno stack di comunicazione, un approccio di integrazione e un'architettura orientata ai servizi (*SoA*) con particolare attenzione alle tecnologie di home automation e building automation.
IoTSyS è stato avviato nell'ambito del progetto di ricerca europeo *IoT6* del *7 ° PQ* (http://iot6.eu/) ed è ora gestito dal Gruppo *Sistemi di Automazione* dell'*Università Tecnologica* di *Vienna*.
Il sistema è sviluppato in *Java* sulla base del *framework OSGi*. La sfida principale è l'ottimizzazione di Internet e delle tecnologie *Web* per consentire operazioni efficienti dal punto di vista energetico all'interno di oggetti intelligenti. Sfrutta gli standard e le tecnologie esistenti come *6LoWPAN*, *CoAP* / *HTTP* e *OBIX*. I contributi principali sono la definizione di un insieme standardizzato di definizione di oggetto basato su *OBIX* per l'*IoT*, un modello di comunicazione *peer-to-peer* basato sul *multicasting IPv6* e il concetto di uno strumento basato sul *Web* che fornisce i mezzi per creare sofisticate logiche di controllo attraverso la programmazione grafica.
Al fine di fornire il livello di astrazione richiesto per un'integrazione unificata e un livello di applicazione comune, *IotSys* utilizza il modello di astrazione della specifica *oBIX* (Open Building Information Xchange). *oBIX* è uno sforzo mirato dei leader del settore e di associazioni di standardizzazione alla creazione di una linea guida standard per *XML* e *Web Services* per facilitare lo scambio di informazioni tra edifici intelligenti, abilitare l'integrazione di applicazioni aziendali e creare vera integrazione di sistemi. Sulla base di standard ampiamente utilizzati dall'industria *IT*, la linea guida *oBIX* cerca di migliorare l'efficacia operativa, consentendo ai gestori di strutture e ai proprietari di edifici di aumentare la conoscenza e il controllo delle loro proprietà. *oBIX* è attualmente un comitato tecnico di *OASIS*. *OASIS* (*Organization for the Advancement of Structured Information Standard*) è un consorzio internazionale senza scopo di lucro che guida lo sviluppo, la convergenza e l'adozione degli standard di e-business. Il consorzio produce più standard di servizi *Web* rispetto a qualsiasi altra organizzazione insieme agli standard per la sicurezza, l'e-business e gli sforzi di standardizzazione nel settore pubblico e per mercati specifici delle applicazioni. Fondata nel 1993, *OASIS* ha oltre 3000 partecipanti che rappresentano oltre 600 organizzazioni e singoli membri in 100 paesi.

oBIX si basa su un'architettura *client / server* orientata ai servizi in cui i *client oBIX* possono accedere ai dati di un server *oBIX* utilizzando i servizi *Web*. Segue più rigorosamente il paradigma di progettazione *RESTful* e fornisce un servizio di lettura, scrittura e recupero da utilizzare per interagire con gli oggetti *oBIX* identificati tramite un *URI* (*Uniform Resource Identifier*). L'obiettivo principale di *oBIX* è fornire un'interfaccia di servizio *Web* aperta per l'accesso a qualsiasi tipo di sistema di automazione degli edifici. Come tecnologia indipendente dalla piattaforma, *oBIX* può essere utilizzato su qualsiasi tecnologia esistente. Di seguito una breve descrizione dei concetti principali definiti dallo standard su cui è fondato *IoTSys*:

- **Oggetti**: Gli elementi di base all'interno di *oBIX* sono *Oggetti*. Ogni *Oggetto* è di un certo *Tipo di oggetto*. La sintassi *XML* viene utilizzata per rappresentare il modello a oggetti. Attualmente sono stati specificati 17 tipi di oggetti standard in cui ogni tipo si associa direttamente a un nome di elemento *XML*. Esempi tipici sono oggetti che modellano elementi di dati singoli come *bool*, *int*, *real* e *string*, nonché tipi di oggetti per rappresentazioni di data e ora.
- **Contratti**: Un concetto importante di *oBIX* è l'uso dei cosiddetti *Contratti*. I *Contratti* sono *Oggetti* standard *oBIX* usati come modello, senza il sovraccarico di introdurre una nuova sintassi: un *Oggetto* può fare riferimento a esso come un "oggetto modello" utilizzando l'attributo “is” nella sintassi *XML*. Vengono utilizzati per definire nuovi tipi di *Oggetto*, ma offrono anche la

possibilità di specificare valori predefiniti. Essi introducono una gerarchia di *Oggetti* . I *Contratti* permettono di risolvere i seguenti problemi:

- **Semantica**: i *contratti* vengono utilizzati per definire i "tipi" all'interno di *oBIX*. Questo consente di concordare collettivamente le definizioni di *oggetti* comuni per fornire una semantica coerente tra le implementazioni dei fornitori. Ad esempio, il *Contratto* Alarm garantisce che il software del client possa estrarre le informazioni sull'allarme in manienra normalizzata dal sistema di qualsiasi fornitore che utilizza la stessa struttura dell'*oggetto*.
- **Predefinizione**: i *contratti* forniscono anche un comodo meccanismo per specificare i valori predefiniti. Si noti che quando si serializzano gli alberi degli oggetti su *XML* (specialmente su una rete), in genere non si consente di utilizzare i valori predefiniti per mantenere l'elaborazione lato client semplice.
- **Esportazione del tipo**: è probabile che molti produttori creino un sistema utilizzando un linguaggio tipizzato in modo statico come *Java* o *C#*. I *contratti* forniscono un meccanismo standard per esportare le informazioni sul tipo in un formato che tutti i *client oBIX* possano comprendere.

- **Operazioni:** Le *Operazioni* sono un tipo di *Oggetto* che definisce l'azione che può essere eseguita su un *Oggetto oBIX*. Tutte le operazioni accettano un *Oggetto* di input come parametro e restituiscono un *Oggetto* come output.
- **Punti:** I *Punti* rappresentano un singolo valore scalare e il suo stato. Tipicamente vengono mappati sensori, attuatori o variabili di configurazione come un insieme di *Punti*, e per catturare una rappresentazione di normalizzazione dei *Punti*, lo standard definisce un'astrazione di basso livello per il *Punto* come un *Contratto* usato per indicare che un *Oggetto* espone la semantica di *Punto*.
- **Cronologia**: Per tenere traccia del valore di un *Punto* nel tempo, lo standard *oBIX* definisce l'*Oggetto* **Cronologia**. Questi *Oggetti* sono composti da un elenco di valori di *Punti* dati con timestamp. Qualsiasi *Oggetto* che desideri esporre se stesso come una cronologia standard di *oBIX* implementa il *Contratto obix:History* (Cronologia).

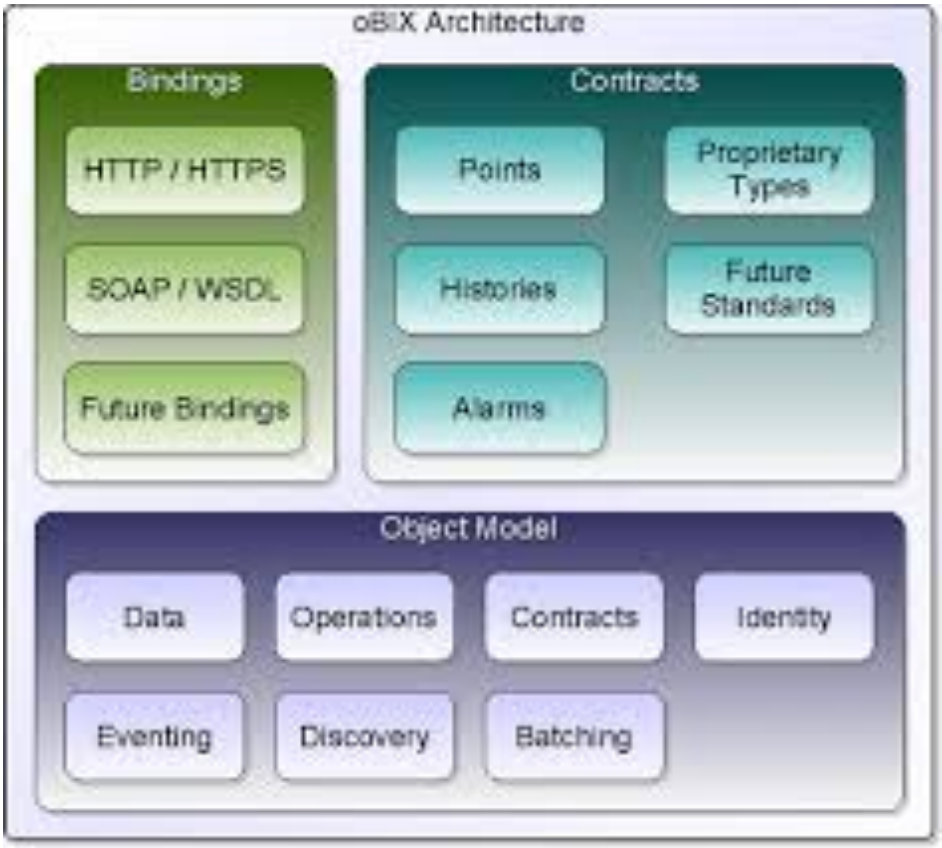

Figura 14: architettura di IoTsys

IoTSys utilizza l'approccio di integrazione centrato sul *Modello Data-point* che, in combinazione con un modello di informazioni generico e il modello di interazione *peer-to-peer*, fornisce interoperabilità senza la necessità di modelli di informazioni specifici sui domini. L'utilizzo di un approccio *Data-point* per l'integrazione significa che ogni dispositivo è espresso come un insieme di *Data-point* di input e output di tipi ben definiti. I *Data-points* sono spesso raggruppati in blocchi funzionali correlati alle capacità desiderate (ad es. un attuatore di interruttore di luce con più canali). Per fare ciò, le specifiche *oBIX* vengono utilizzate per fornire una base per uno *stack IoT*. In particolare, le specifiche *oBIX* definiscono due tipi di oggetti, *osservatori* e *allarmi*, utilizzati per definire le regole di automazione. *oBIX* fornisce un modello per il client per accedere a informazioni quasi in tempo reale di ogni dato o *Oggetto* presente nel sistema utilizzando tecniche di polling: gli *osservatori*. Dopo che un oggetto *osservatore* è stato creato sul *WatchService* del *server*, ad esempio utilizzando l'*API REST* dell'interazione http esposta da *IoTsys*, il *client* può registrare un nuovo *Oggetto* da monitorare chiamando l'operazione *add* sull'*Oggetto osservatore*. Per altri tipi di operazioni che possono essere eseguite con gli *osservatori*, fare riferimento alle specifiche *oBIX* .
Un'altra funzionalità definita da *oBIX* è la capacità di generare *Oggetti allarme*: un *allarme* notifica una condizione che richiede di essere inviata ad un utente o ad un'altra applicazione. Il meccanismo di *allarme* consiste in un algoritmo lato server che monitora una fonte di allarme, cioè i sensori (questa stanza è troppo calda), i problemi hardware (il disco è pieno) o le applicazioni (l'edificio sta consumando troppa energia). Se viene rilevata una condizione di allarme, viene generata una registrazione di allarme descritta come "non normale". Se questa condizione ritorna normale, può essere generato un altro evento di allarme.

Per gestire e manipolare la costruzione della logica di controllo in modalità *drag and drop*, *IoTsys* fornisce una *GUI* basata su *AngularJS* denominata *oBeliX*.

HomeGenie

HomeGenie è un framework di interoperabilità domotica multi-piattaforma che, diversamente da molti altri, è sviluppato in *C#* utilizzando il framework *MONO* per la distribuzione *UNIX*. Quindi può funzionare sia su *computer embedded* che desktop (*Windows*, *Linux* e *Mac*).
Il framework è sviluppato da *G-Labs*, una fabbrica di codice *open source*. Il numero di sviluppatori coinvolti in *homeGenie* è di circa 20. Il framework è rilasciato in versione beta.

Il nucleo di *HomeGenie* è composto da un motore di automazione generico chiamato *A.P.E.*, il *Automation Programs' Engine*, in grado di integrare diversi *bus* o *LAN* domotiche e visualizzare i dati in un'interfaccia grafica personalizzabile. Il programma di automazione può essere sviluppato in cima alla *A.P.E.* e reso disponibile agli utenti tramite il gestore di pacchetti integrato. Grazie alla *GUI*, il sistema può essere utilizzato anche da utenti meno esperti, ma con limitazioni nello sfruttamento del suo potenziale, mentre uno sviluppatore può sfruttare lo strumento di programmazione impostato da *HomeGenie*, che consente l'uso di linguaggi diversi come *C#*, *python* , *javascript* o *ruby*, per sviluppare regole di automazione complesse o manager.

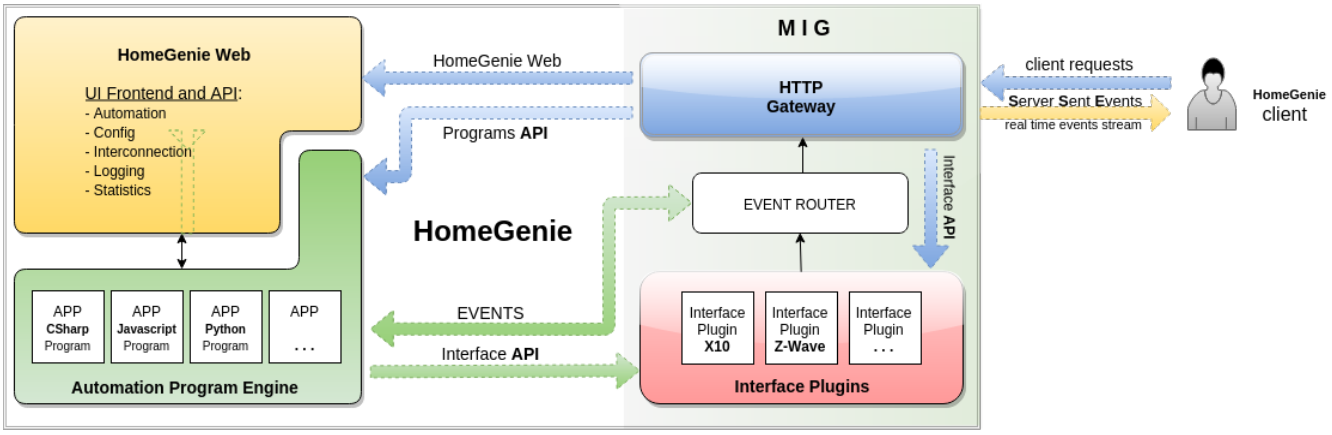

Figura 15: Architettura di HomeGenie

In *HomeGenie* l'astrazione di dispositivi e servizi viene implementata tramite entità chiamate *moduli virtuali*. Per identificare ciascun modulo in un modo unico, il nome è composto da:

- **Dominio**, che raggruppa moduli della stessa classe di solito in relazione con la tecnologia utilizzata da esso, come ad esempio *HomeAutomation.X10* o *HomeAutomation.ZWave*;
- **Indirizzo,** che è un *ID* univoco necessario per identificare ogni modulo nella stessa classe
- **Tipo,** che definisce la categoria del dispositivo o servizio rappresentato dal modulo, ad esempio Interruttore, Luce, Dimmer, Sensore, Temperatura, Ventola, Termostato, Finestra, Porta, ecc..

Ogni *modulo* ha un set di campi di parametri che rappresentano i dati utili a descrivere lo stato del dispositivo o servizio associato, ad esempio *Status.Level* per lo stato corrente di una luce. I moduli virtuali possono essere installati utilizzando il gestore pacchetti integrato.
Altri componenti di *HomeGenie* sono il *Modulo Programma*, che è un programma di automazione e *Widget*, utilizzato per visualizzare i dati dei moduli.

HomeGenie è un sistema basato su eventi e fornisce diversi modi per creare l'interoperabilità tra i dispositivi. Il modo più semplice è utilizzare il *modulo* già disponibile nel sistema e utilizzare *WebGui* per configurare alcune regole tra i componenti. Il modo più espressivo, invece, è sviluppare un *Modulo Programma*. Il sistema fornisce un editor integrato nel *WebGui* e una serie di *HelperFunctions* per interagire direttamente con il sistema. Con questa funzione è possibile associare un programma a un *modulo virtuale*:

```
Program.AddVirtualModule(
  "<module_domain>",          // <-- nome di dominio
  "<module_address>",         // <-- indirizzo del modulo
  "<module_type>",            // <-- tipo del modulo
  "homegenie/generic/switch"); // <-- widget usato per mostrare il modulo a video
```

HomeGenie helperFunction fornisce anche una serie di metodi per interrogare i dispositivi in base a tipo, dominio, gruppo o tipo di funzionalità associati a un dispositivo, ma la mancanza di un'ontologia o tassonomia ben definita influenza il riutilizzo del codice su diverse installazioni del sistema gestito da diverso utente. *HomeGenie* è in grado di inoltrare gli eventi ad altre istanze del sistema. I programmi di automazione possono ricevere eventi dal modulo locale o remoto e servizi esterni e reagire generando altri segnali che vengono quindi elaborati e propagati attraverso il sistema.

The ThingSystem

The *ThingSystem* è un framework di interoperabilità domotica composto da un insieme di componenti software e protocolli di rete per l'*IoT* con particolare attenzione per il sistema di automazione domestica. Il sistema principale è open source ed è composto da un manager centrale, chiamato *steward*, che contiene diversi driver per prodotti reali. La direttiva principale dello *steward* è quella di permettere una forma di interoperabilità tra le *LAN* domotiche collegate al cuore del framework, definendo un modello di attività (le regole che l'amministratore applica), insieme a una serie di interfacce per le terze parti.
Gli *attori* sono i soggetti identificati dal framework per modellare le entità coinvolte nel sistema domotico. Soprattutto un *attore* fa riferimento a un prototipo di un'entità che partecipa a un'attività di automazione. Ci sono tre tipi di *attori*:

- **Dispositivi:** sono i dispositivi fisici domotici attaccati alle varie reti o *LAN* domotiche, come ad esempio attuatori, luci, termostati, ecc…
- **Gruppi:** combinano gli attori in insiemi che hanno una qualche relazione logica fra loro
- **Pseudoattori:** sono costrutti solo software come, ad esempio, la rappresentazione dei luoghi o stanze di una casa

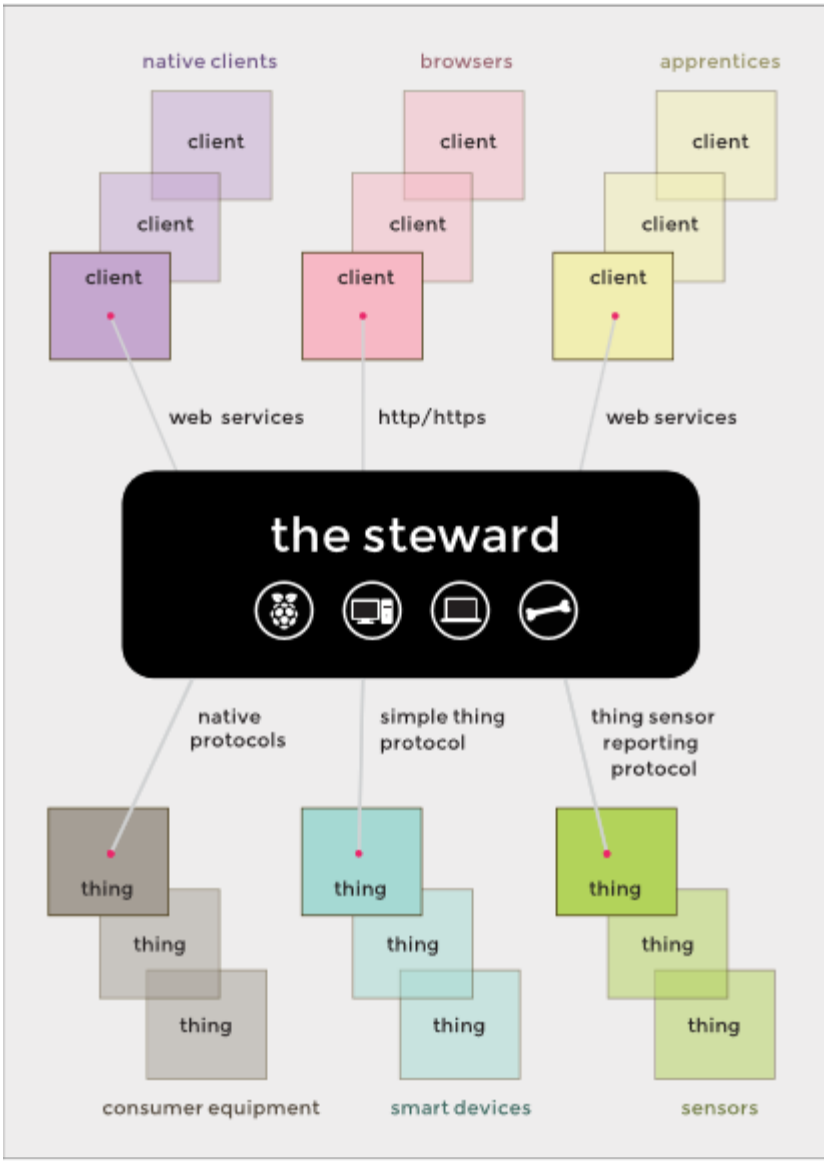

Figura 16: Architettura di The ThingSystem

Nel modello del framework, i *dispositivi* hanno tre proprietà obbligatorie di base: *nome*, *stato* e *aggiornamento*. A seconda del tipo a cui appartengono, hanno anche proprietà aggiuntive. I dispositivi ereditano da una o più (eredità multipla) delle seguenti 10 categorie: *clima, gateway, indicatore, illuminazione, media, movente, presenza, sensore, interruttore* e *indossabile*. The *ThingSystem*, come altri framework, definisce una “lingua franca” o tassonomia, consistente in una gerarchia di prototipi di dispositivi e una lista di proprietà. La denominazione gerarchica viene utilizzata per i prototipi di dispositivo. Il modello non impone rigidi vincoli alle proprietà: possono apparire in qualsiasi prototipo di dispositivo in cui "ha senso".

Il supporto per un nuovo dispositivo viene aggiunto creando un modulo con un nome di percorso che rispetti la tassonomia definita dal framework nel seguente modo:

/device/presence/maker/arduinoPirSensor

In questo caso la prime due parti del nome indicano che questa cosa è un dispositivo che segnala la presenza di una persona. Le proprietà di questo tipo di dispositivi sono definite dalla tassonomia; in questo caso, ad esempio, sono obbligatorie due proprietà: *stato* dei dispositivi (presente o assente) e *lqi*, un valore intero che indica la magnitudine di prossimità.
Quando il modulo viene caricato, durante l'avvio, viene creato un *oggetto* che viene consultato quando viene rilevata un'istanza del dispositivo e che definisce in che modo vengono rilevate le istanze del dispositivo. Può contenere due sezioni opzionali: *osservare* che implementa l'osservazione asincrona degli eventi ed *esegue* che implementa l'esecuzione delle attività.
Esistono tre tipi di attori che rappresentano dispositivi:

- **Attore indipendente**: fa riferimento a un dispositivo scoperto dallo *steward* e che non ha capacità di discovery
- **Gateway e attore subordinato**: in genere lo *steward* è in grado di scoprire un gateway per una particolare tecnologia. Anche in The *ThingSystem*, i gateways sono moduli software che fanno da "traduttori" per le varie *LAN* domotiche. Quando un attore gateway scopre un dispositivo , chiama un metodo per dire allo *steward* di (ri) crearlo.
- **Attore creabile**: in generale, questi attori fanno riferimento a costrutti solo software: mentre è compito dello *steward* di scoprire i dispositivi, solo un utente può decidere se desidera che le letture dei sensori vengano caricate da qualche parte. Questo tipo di attori non è rilevabile, quindi un cliente deve effettuare una chiamata *API* specifica allo *steward* per creare un'istanza.

Per aggiungere il supporto per un nuovo dispositivo, il framework fornisce due tipi di protocolli di comunicazione basati sul requisito del dispositivo. Il *Thing Sensor Reporting Protocol* (*TSRP*) è un semplice protocollo multicast basato su *UDP* che può essere utilizzato se un dispositivo deve solo segnalare la lettura di un sensore o un evento. Per i dispositivi più sofisticati in grado di eseguire un'attività e, infine, di segnalare una misura o un evento, è necessario utilizzare il protocollo *SimpleThing*, basato su *websocket*. Ciò consente allo *steward* di chiedere al dispositivo di eseguire un'azione invece di aspettare solo un messaggio in arrivo. I protocolli supportati sono già molti, come ad esempio *Nest*, *Netatmo*, *Hue*, *Mqtt*, *Z-wave switch* e così via.
L'nteroperabilità tra dispositivi in The *ThingSystem* si basa sui concetti di *evento, task, gruppi* e *attività.*
Gli *eventi* sono un concetto chiave del *Thing System*. Lo *steward* genera un *evento* quando riceve una richiesta da un dispositivo che ad esempio utilizza il *Thing Sensor Reporting Protocol*. Un evento è formato principalmente da quattro campi:

- Un *eventID* fisso assegnato dallo *steward* durante la creazione dell'*evento* e utilizzato da ogni successiva chiamata *API* che fa riferimento a tale *evento*.
- Un *eventUID* fisso assegnato dal client durante la creazione dell'*event*o e utilizzato per garantire l'atomicità della creazione dell'*evento*.
- Un *actorType* ("dispositivo", "luogo" o "gruppo") e un *actorID* assegnati dal client durante la creazione dell'*evento*.
- Un *verbo osservare*: Il valore del parametro di questo campo è specifico per l'attore e viene assegnato dal client durante la creazione dell'*evento*.

Un *task* consiste in una definizione di un'azione da eseguire. Un *task* è principalmente descritto da quattro campi:

- Un *taskID* fisso assegnato dallo *steward* durante la creazione del *task* e utilizzato da qualsiasi successiva chiamata API che fa riferimento a tale *task*.
- Un *taskUID* fisso assegnato dal client durante la creazione del *task* e utilizzato per garantire l'atomicità della creazione del *task*.
- Un *actorType* ("dispositivo", "luogo" o "gruppo") e un *actorID* assegnati dal client durante la creazione del *task.*
- Un *verbo di esecuzione* e un valore di parametro specifici per l'attore che viene assegnato dal client durante la creazione del *task*.

L'astrazione del *gruppo* può essere utilizzata per aggregare attori (ad esempio dispositivi dello stesso tipo, come luci) e utilizzarli come input o obiettivi delle regole di automazione.
Tutti i concetti precedenti sono definiti per essere combinati dalle *attività*. Un'*attività* è un vincolo tra un *evento* e un *task* o un *gruppo* di essi. L'architettura di un'*attività* consiste principalmente di:

- Un *activityID* fisso assegnato dallo *steward* durante la creazione dell'*attività* e utilizzato da qualsiasi successiva chiamata *API* che fa riferimento a tale *attività*.
- Un *activityUID* fisso assegnato dal client durante la creazione *dell'attività e* utilizzato per garantire l'atomicità della creazione dell'*attività*.
- Un *eventType* ("dispositivo", "luogo" o "gruppo") e un *eventID* assegnati dal client durante la creazione dell'*attività.*
- Un *taskType* ("dispositivo", "luogo" o "gruppo") e un *taskID* assegnati dal client durante la creazione dell'*attività.*

Il ciclo di osservazione e esecuzione del framework è responsabile di osservare l'*evento* ed eseguire il *task* associato mediante l'*attività*.

Architettura SHELL

La Figura 17 rappresenta, in maniera semplificata ma completa il modello generale ideato per il framework *SHELL*. Il modello ideato si basa sul paradigma degli *oggetti* fisici e/o logici (*OBJECTS*) interagenti fra loro mediante una rete interna (*I-NET*) e localizzati in un ambiente opportuno (*SPACE*) in grado di offrire dei servizi generali agli oggetti (*SERVICES*) ed in grado di comunicare con il mondo esterno (*WORLD*) mediante una rete esterna (*E-NET*). Gli utenti (*USERS*) possono interagire con il sistema basato sul framework *SHELL* attraverso uno o più oggetti specifici dotate di opportune interfacce uomo-macchina (*HMI*). Gli *oggetti*, fisici o logici, possono essere raggruppati in gruppi (*GROUPS*), anch'essi fisici o logici, caratterizzati dalla presenza di un *oggetto* denominato *manager* (*MANAGER*) che gestisce il gruppo e ne determina le funzionalità e lo scopo. Un determinato *oggetto* può essere presente contemporaneamente in più gruppi. Un gruppo composto da soli *oggetti manager* è un gruppo di secondo livello (*CLUSTER*) cioè un gruppo che riunisce esclusivamente altri gruppi. Nell'architettura, all'interno dello *SPACE*, si trovano gli *oggetti*. Un *oggetto* rappresenta un insieme di funzionalità coerente con un servizio specifico e può appartenere a due categorie:

- **Oggetto reale**: rappresenta un *oggetto* del mondo reale, con associato uno specifico hardware, che realizza un insieme di funzionalità coerente con un servizio tangibile (ad esempio: una centralina metereologica con hardware installato sul posto).

- **Oggetto virtuale**: rappresenta un *oggetto* logico, quindi senza uno specifico hardware, che realizza un insieme di funzionalità coerente con un servizio non tangibile, tipicamente un algoritmo (ad esempio, una centralina metereologica virtuale che rielabora e presenta i dati metereologici trovati in Internet, non c'è nessun hardware locale).

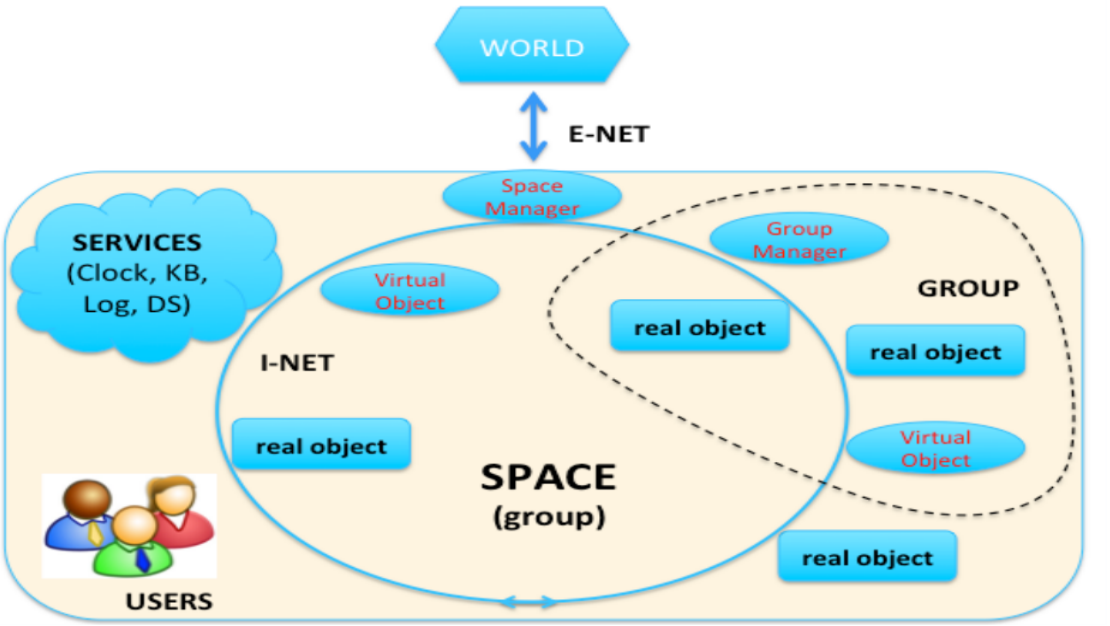

Figura 17: architettura SHELL

Ogni *oggetto* possiede un identificatore (*OBJECT_ID*) univocamente definito sia nello *SPACE* che nel *WORLD* ed uno stato (*OBJECT_STATUS*) che descrive lo stato dell'oggetto nella *I-NET* (ad esempio: richiesta di installazione, attivo, non attivo disabilitato, etc.).

Ogni *oggetto* realizza delle funzionalità opportunamente definite, per esempio in termini di comandi eseguibili, segnali generabili, ecc. A ogni funzionalità può essere collegata una lista di regole (*F-ACL*) per stabilire chi può utilizzarla (ad esempio: un altro oggetto generico e/o un manager, etc.) e con quale *USERLEVEL*, se la richiesta proviene da un oggetto con *HMI*. A questo scopo i dati scambiati possono essere criptati e/o autenticati, se necessario. Gli oggetti interagiscono fra loro esclusivamente mediante comandi e segnali, anche se all'interno di un'interazione questi ultimi possono gestire altre modalità temporanee di scambio dati (es. canali dedicati per audio/video streaming). Le funzionalità di un dispositivo possono essere raggruppate in classi di funzionalità (*F-CLASS*). Ogni classe rappresenta un insieme di funzionalità coerente con la definizione di un servizio tipico, e costituisce l'insieme minimo e generale di comandi e segnali che possono logicamente descrivere tale servizio. Per esempio nel caso di servizio "lavatrice", la relativa classe di funzionalità descriverà il funzionamento base dell'oggetto, cioè tutti i comandi e segnali che sono comuni a tutte le lavatrici e che permettono quindi di identificare univocamente le funzionalità di una generica lavatrice.

Eventuali funzionalità specifiche di un modello ma non di altri non fanno parte della classe di funzionalità lavatrice ma saranno comprese fra le funzionalità dell'oggetto specifico (cioè comandi e segnali specifici di quel solo oggetto). Ogni classe di funzionalità ha un identificatore unico nello *SPACE* e nel *WORLD* (*F-CLASS_ID*) e può dipendere dalla presenza di altre classi di funzionalità. Tali dipendenze (*F-CLASS_DEPENDENCES*) fanno parte della definizione di una classe di funzionalità insieme ai comandi e ai segnali accettati e generati. In generale, un oggetto può essere definito dalla composizione di una o più *F-CLASS* e delle relative dipendenze e da comandi e segnali aggiuntivi specifici per quell'oggetto. Questo permette di garantire da una parte la generalità (tutti gli oggetti di un certo tipo devono comportarsi nello stesso modo) attraverso l'uso delle *F-CLASS* e dall'altra la specificità degli oggetti (l'oggetto di un particolare *Brand* deve poter esporre delle funzionalità avanzate e *vendor-specific*) attraverso l'uso dei comandi e segnali specifici dell'oggetto.

La classe di funzionalità *OBJECT* deve essere sempre definita e deve raggruppare tutte le funzionalità base comuni a tutti gli oggetti del sistema (ad esempio: i comandi e segnali per gestire la fase di installazione e configurazione dell'oggetto).

Le classi di funzionalità (*F-CLASSES*) e i relativi comandi e segnali rappresentano la base concettuale per definire una "lingua franca" indipendente dai dispositivi fisici o logici collegati al framework: questa lingua franca e il relativo significato semantico, che sarà contenuto in una apposita *KnowledgeBase* (Base di conoscenza), vengono utilizzati dal framework per definire il formato dei messaggi interni al framework e servono per implementare un livello superiore di interoperabilità semantica oltre che sintattica tra dispositivi eterogenei esterni, collegati alla *E-NET*, come ad esempio le reti domotiche già presenti nella *Smart Home*: esempio di ciò sono le reti *KNX* e *BTicino* descritte successivamente in questo documento.

Le *E-NET* si connettono al bus interno *I-NET* del framework *SHELL* attraverso dei gateways software che nel caso del framework *SHELL* sono chiamati *drivers*. Attraverso i gateways, i dispositivi domotici saranno

funzionalmente integrati e controllati dal framework *SHELL*. In questo modo, il framework sarà in grado di creare un livello logico di astrazione dei dispositivi in modo da eliminare tutte le loro differenze tecnologiche, permettendo di gestirli indipendentemente dalle loro peculiarità hardware e software.

Visto che ogni tecnologia domotica ha le proprie caratteristiche e modelli di funzionamento che li rende incompatibili con gli altri, occorre che il gateway suddetto sia costruito su misura, ma è sufficiente costruirne uno per tecnologia (Figura 18). Il compito principale di un driver *SHELL* è quello di convertire gli eventi e i comandi ricevuti dalle varie *LAN* nella "lingua franca" di *SHELL* laddove è possibile, e viceversa. Ogni driver in *SHELL* quindi gestirà un insieme di dispositivi che faranno parte di una stessa *LAN* ma che vengono visti ognuno separatamente. Le loro funzionalità peculiari e i comandi e segnali *SHELL* supportati dal dispositivo particolare verranno descritti in un file di configurazione apposito insieme ai dati necessari per identificare univocamente il dispositivo in *SHELL*, in modo da mapparlo con un dispositivo logico virtuale interno al framework e di poter indirizzare i dispositivi collegati alle *LAN* convertendo il meccanismo di indirizzamento di *SHELL*, basato su un *ID* univoco di tipo stringa, in quello proprietario della *LAN*. Nello stesso file di configurazione, saranno riportate le regole e le modalità di conversione del comando e della eventuale risposta, così come degli eventi ricevuti dalla *LAN* da convertire in segnali *SHELL*.

In generale, ogni driver dovrà essere in grado di:

- ottenere la lista dei dispositivi domotici appartenenti ad una determinata tecnologia da includere all'interno dell'istanza del framework;
- creare all'interno del framework un livello di astrazione logico per i dispositivi individuati;
- effettuare un'associazione tra i dispositivi logici creati all'interno del framework ed i dispositivi domotici reali;
- effettuare un'associazione tra funzionalità e comandi definiti per i dispositivi logici e le funzionalità e comandi dei dispositivi domotici reali, secondo le tassonomie definite in *SHELL*;
- effettuare un'associazione tra tipi di dato e valori definiti in *SHELL* e tipi di dato e valori propri della tecnologia domotica;
- effettuare la connessione al canale fisico della rete domotica;
- ricevere dal framework comandi in formato *SHELL* provenienti dalla *business logic* e convertirli in comandi comprensibili al dispositivo domotico destinatario ed inviare la relativa risposta o un feedback dopo l'esecuzione;
- catturare gli eventi che vengono propagati all'interno della rete domotica relativi ai dispositivi di interesse e convertirli in eventi in formato *SHELL* per la loro propagazione all'interno della piattaforma (*business logic*).

Queste mappature e conversioni vengono effettuate dalla implementazione interna al *driver* e sono completamente trasparenti alle due *LAN* che il *driver* mette in interoperabilità: il *driver* infatti risponderà ai comandi *SHELL* secondo le modalità e la semantica e sintassi delle risposte a questi ultimi come definite nella *KnowledgeBase*, e alla *LAN* verranno mandati messaggi nel suo formato particolare e secondo le modalità necessarie. Viceversa, le risposte o gli eventi (come ad esempio, la pressione di uno switch o la rotazione di una manopola per la regolazione della temperatura di un termostato) saranno convertiti in un corrispondente messaggio *SHELL* come definito nella *KnowledgeBase*, secondo le modalità di conversione riportate nel file di configurazione o implementate direttamente nel *driver*,

così come anche le differenze e le conversioni tra i tipi dei parametri dei messaggi *SHELL* e gli eventuali valori degli stati dei dispositivi secondo il formato richiesto dalla tecnologia domotica.
Nei prossimi due paragrafi, sono riportati gli esempi e il funzionamento, ad alto livello, di due *LAN* e tecnologie domotiche che verranno implementati in *SHELL* come *drivers*: *BTicino* e *KNX*.

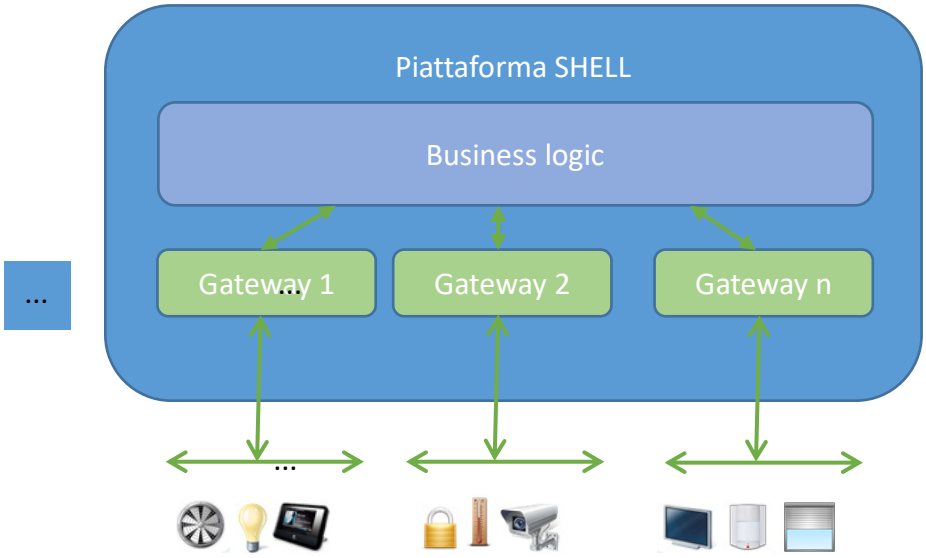

Figura 18: I gateways nell'architettura SHELL

Architettura KNX

KNX è uno standard europeo (*EN 50090 - EN 13321-1*) e mondiale (*ISO/IEC 14543*) di *building automation* aperto e coperto da royalty. *KNX* è sviluppato da *KNX Association* sulla base dell'esperienza dei suoi predecessori *BatiBUS*, *EIB* e *EHS*. I membri sono costruttori che sviluppano dispositivi basati su *KNX* per diverse applicazioni nel controllo di case ed edifici e nella integrazione delle loro funzionalità come illuminazione, tapparelle, riscaldamento, ventilazione, aria condizionata, gestione energia, contabilizzazione, monitoraggio, sistemi di allarme/antintrusione, elettrodomestici, audio/video e molto altro. L'obiettivo comune di queste aziende è in generale di promuovere lo sviluppo dei sistemi di installazione per gli edifici e *KNX* come standard mondiale aperto per il controllo delle case e degli edifici.

Lo standard *KNX* prevede diversi mezzi trasmissivi che possono essere utilizzati in combinazione, tra cui il *twisted pair*, *power line*, *radio frequency* ed *ethernet*. Per collegare il dispositivo al *bus*, è necessaria un'interfaccia, costituita da un componente elettronico detto *BCU* (*Bus Coupling Unit*). Il *BCU* interpreta i pacchetti ricevuti dal *bus*, li traduce nei segnali adatti al dispositivo, e viceversa. Ogni *BCU* possiede un indirizzo, non necessariamente univoco, che lo identifica; il meccanismo d'indirizzamento e di trasmissione differisce a seconda degli standard.

La struttura di un sistema *BUS KNX* è composta da aree e linee (Figura 19). Le aree possono essere fino a un massimo di 15, collegate tra loro da una linea dorsale principale. In ciascuna area vi è una linea principale, da cui possono svilupparsi diverse linee secondarie, fino a un massimo di 15 linee. Sulle linee secondarie sono collegati i dispositivi *KNX* (sensori, attuatori) e il numero massimo di dispositivi all'interno di ogni linea è di 256 suddivisi in 4 segmenti di linea da 64 dispositivi.

Gli accoppiatori svolgono una funzione molto importante in un sistema *BUS*. Essi infatti provvedono a isolare elettricamente le varie parti del sistema in modo da evitare che un guasto elettrico di un singolo dispositivo comprometta la funzionalità dell'intero sistema. Inoltre, l'accoppiatore funziona anche come filtro sui messaggi che sono trasmessi dai singoli dispositivi, in modo da evitare che questi vengano trasmessi su tutta la rete.

Alla base del concetto di applicazione proposto da *KNX* c'è il concetto dei *datapoint*. Con questo termine vengono indicate tutte le variabili di processione e controllo dell'intero sistema. I *datapoint* possono costituire input, output, parametri, dati di tipo diagnostico, etc. e sono "contenuti" in *Group Objects* e in *Interface Object Properties*. In una rete *KNX* ogni dispositivo deve avere un indirizzo univoco, chiamato indirizzo individuale. Nello standard *KNX* il trasferimento dell'informazione tra dispositivi avviene tramite strutture dati dette "*datapoint*", il termine "*datapoint*" si può considerare sinonimo di variabile condivisa. Il collegamento si realizza assegnando a ogni *datapoint* un codice numerico detto "indirizzo di gruppo", del tutto indipendente dal valore dell'indirizzo del dispositivo.

Per "condividere" un insieme di variabili tra più dispositivi è necessario, quindi, che esse abbiano lo stesso indirizzo di gruppo e siano dello stesso tipo (*bit*, *byte*, *word*, ecc.). Questo modalità è denominata *System mode* e di fatto realizza il "*free binding*" tra le variabili dei vari dispositivi.
La trasmissione dei dati è di tipo *multicast*: il nodo che ha un determinato *datapoint* in uscita (*flag* di tipo trasmissione attivato) invia in rete un pacchetto che sarà ricevuto contemporaneamente da tutti i dispositivi che hanno, in ingresso, un *datapoint* dello stesso tipo, avente lo stesso indirizzo di gruppo di quello mittente.

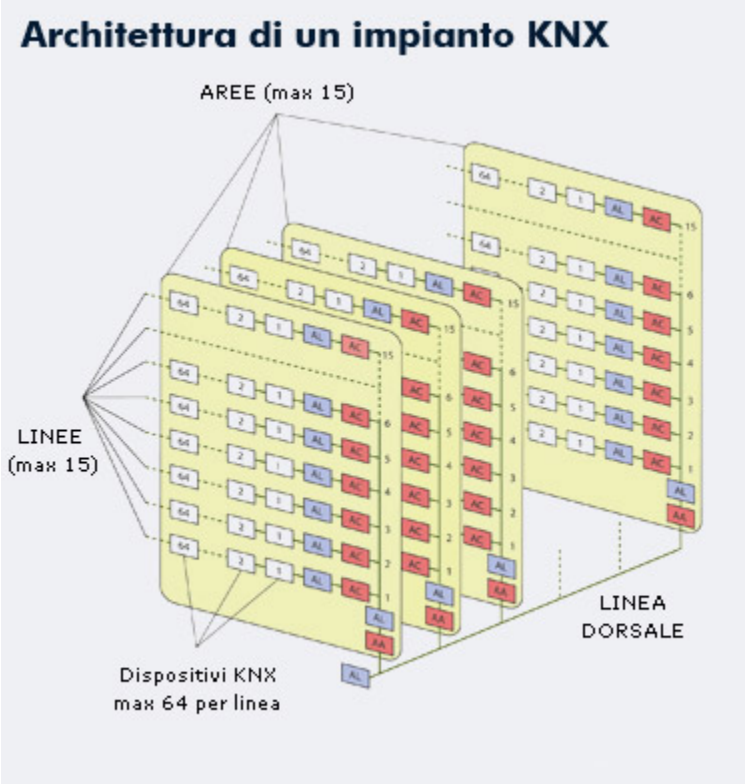

Figura 19: architettura KNX

Architettura MyHome (BTicino)

La ***BTicino S.p.A.*** è un'azienda italiana che opera nel settore delle apparecchiature elettriche in bassa tensione destinate agli spazi abitativi, di lavoro e di produzione, distinte in soluzioni per la distribuzione dell'energia, per la comunicazione (citofonia, videocitofonia, dati) e per il controllo di luce, audio, clima e sicurezza. L'azienda fa parte del gruppo internazionale *Legrand* e opera sia sul mercato italiano sia su quello mondiale con oltre 60 sedi all'estero.

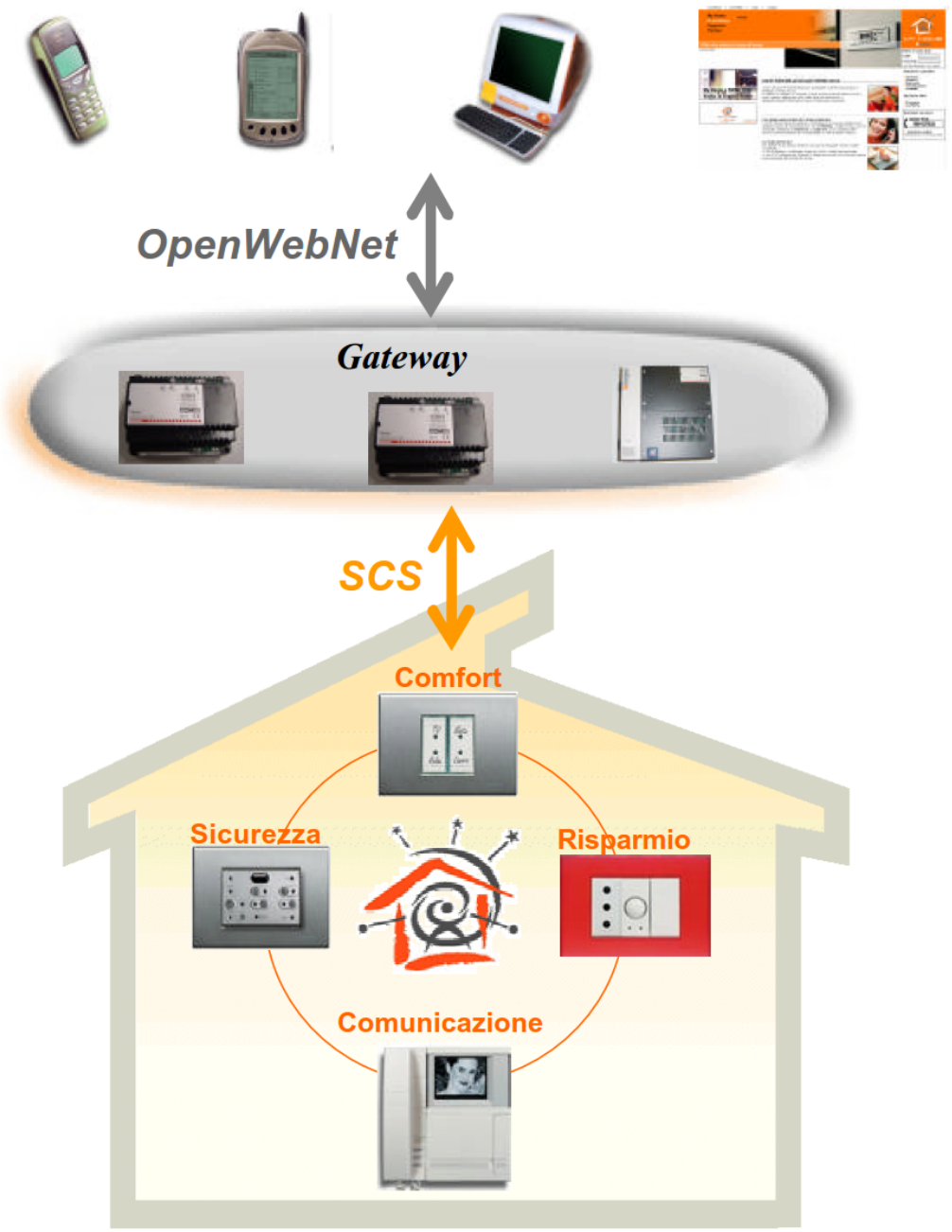

Figura 20: architettura MyHome

La rete domotica proprietaria *MyHome* prevede l'uso di un bus su filo chiamato *SCS* che collega i vari dispositivi, tra cui lampadine intelligenti, tapparelle, termostati, videocamere di sorveglianza, sistemi multimediali di intrattenimento ecc… divisi in ambienti.

La struttura di un sistema *bus SCS* è composta da più livelli, collegati da interfacce: ogni livello può indirizzare fisicamente fino a 256 dispositivi utilizzando valori numerici, mentre i livelli possono essere a sua volta illimitati, in quanto si può indirizzare 256 ulteriori livelli, ognuno a sua volta diviso in 256 livelli e così via.

Per poter accedere al *bus SCS* e ai dispositivi collegati ad esso dall'esterno, BTicino ha sviluppato un apposito *gateway hardware* (modelli *F453* e successivi) raggiungibile tramite rete *Ethernet* e indirizzo *IPv4* programmabile: il *gateway* si occupa di interpretare comandi in un formato aperto sviluppato appositamente da *BTicino* chiamato *OpenWebNet*, ricevendoli dai dispositivi esterni o interni al *bus SCS*, instradarli al dispositivo destinatario sul *bus SCS* e di recapitare le eventuali risposte, o gli eventi generati dai dispostivi collegati sul *bus SCS* come la pressione di interruttori bifase, la manopola di un termostato, ecc…

Secondo le specifiche *MyHome*, *OpenWebNet* è un protocollo pensato per essere indipendente dal mezzo di comunicazione utilizzato. Un messaggio *OpenWebNet* è caratterizzato da una struttura con campi a lunghezza variabile separati dal carattere speciale '*' e chiuso con '##'. Sono previste stringhe per l'invio di comandi, la richiesta dello stato dei dispositivi/sistemi, la scrittura e la lettura di grandezze.
Un messaggio *OWN* è composto da caratteri appartenenti solo e soltanto al seguente insieme *Sym* = **{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *, #}.** Un messaggio *OWN* sintatticamente corretto inizia con il carattere '*', e finisce con la coppia di caratteri '##'. Il carattere '*' viene anche usato come separatore dei *tag*. Un messaggio *OWN* è così strutturato: ***tag1*tag2*tag3*...*tagN##** dove un tag è composto da caratteri appartenenti all'insieme *Sym*. Un *tag* non può contenere la coppia di caratteri '##'. Un *tag* può essere anche omesso, creando così messaggi open del tipo ***tag1***tag4*...*tagN**##**

In Figura 21 è riportato il formato sintattico dei principali comandi *OWN*.

ACK	*#*1##
NACK	*#*0##
NORMALE	*CHI*COSA*DOVE#LIV#INT##
RICHIESTE STATO	*#CHI*DOVE#LIV#INT##
RICHIESTA GRANDEZZA	*#CHI*DOVE#LIV#INT*GRANDEZZA##
SCRITTURA GRANDEZZA	*#CHI*DOVE#LIV#INT*#GRANDEZZA*VAL1*VAL2*...*VALn##

Figura 21: formato comandi OWN